

PEARSON
Prentice
Hall

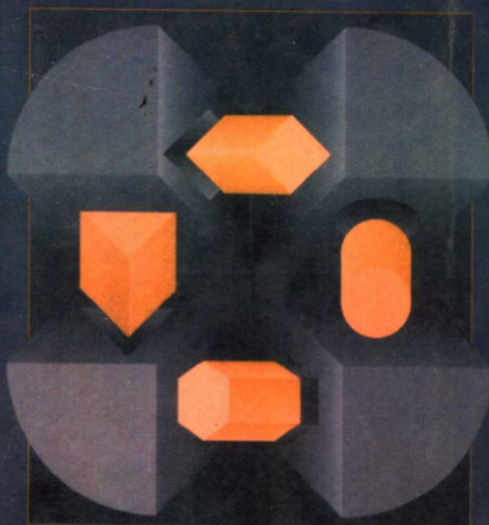
计 算 机 科 学 丛 书

嵌入式系统的 描述与设计

(美) Daniel D. Gajski Frank Vahid Sanjiv Narayan Jie Gong 著

边计年 吴为民 等译
清华大学

SPECIFICATION
AND DESIGN
OF EMBEDDED
SYSTEMS



DANIEL D. GAJSKI / FRANK VAHID
SANJIV NARAYAN / JIE GONG

Specification and Design of
Embedded Systems



机械工业出版社
China Machine Press



在过去的十年里，VLSI设计技术，特别是CAD产业，以异常迅猛的速度发展。这种快速的进展使得产业界能在更短的时间内设计和制造出复杂的专用集成电路和系统。本书综述了系统设计的基本概念，并提出了软件和硬件系统设计方法学的原理。可帮助投身于电子系统设计的人士跟上时代的步伐。

本书特点

- 提出系统设计中的基本问题，讨论各种可用于捕获系统行为及其实现的概念模型
- 研究用来描述系统功能性的语言，以及通过模拟验证系统功能的各种问题
- 为系统划分、评估及模型细化等提供了算法和技术的综述
- 将以上所有的主题结合起来，使之成为一个具有一致性的设计方法，其中还包括对系统设计的通用环境的讨论

主要作者简介

Daniel D. Gajski 教授于宾夕法尼亚大学费城分校获得博士学位。他有十几年的从事数字电路、交换系统、巨型计算机设计以及在VLSI结构领域的从业经历。此后又在伊利诺伊大学厄巴纳-尚佩恩分校计算机科学系从事了10年的学术研究。目前他是加州大学艾尔温分校信息与计算机科学系、电子与计算机工程系的教授。他出版过多部专著。他的研究方向为嵌入式系统与信息技术、设计方法学与e设计环境、系统描述语言及CAD软件、设计科学。

主要译者简介

边计年 清华大学计算机系教授，博士生导师。1970年毕业于清华大学自动控制系，毕业后在清华大学任教至今。他出版过多部专著和译著。研究方向为片上系统(SOC)的系统设计方法，包括系统描述、软硬件划分与通信综合、与布图结合的高层次综合、系统协同设计与验证等。

PEARSON
Prentice
Hall

www.PearsonEd.com

ISBN 7-111-16422-9



9 787111 164227

封面设计：陈子平



华章图书

华章网站 <http://www.hzbook.com>

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzsj@hzbook.com

ISBN 7-111-16422-9/TP · 4270

定价：33.00 元

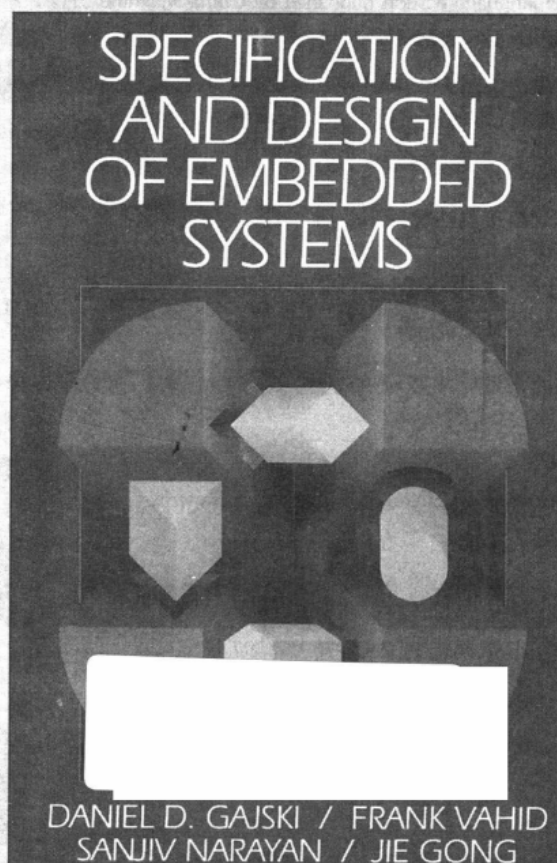
计 算 机 科 学 丛 书

嵌入式系统的 描述与设计

TP360.21
14

(美) Daniel D. Gajski Frank Vahid Sanjiv Narayan Jie Gong 著

边计年 吴为民 等译
清华大学



Specification and Design of Embedded Systems



机械工业出版社
China Machine Press

本书介绍嵌入式系统领域的基本概念以及实际的描述和 design 方法,包括嵌入式系统设计的模型和体系结构、描述语言、系统划分、设计质量评估、描述细化以及系统级方法学等方面。讲解详细,实例丰富,有针对性地介绍了若干著名算法或解法,并解释每种方法的优缺点,还包括对该领域其他工作的综述,并提出尚未解决的一些问题。本书适合从事嵌入式系统设计和研究的工程技术人员、科研人员、高等院校计算机和电子信息工程专业的本科生和研究生。

Authorized translation from the English language edition entitled *Specification and Design of Embedded Systems* by Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong, published by Pearson Education, Inc., publishing as Prentice-Hall (ISBN 0-13-150731-1), Copyright © 1994 by Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanic, including photocopying, recording, or by any information storage retrieval system, without permission of Pearson Education, Inc.

Chinese Simplified language edition published by China Machine Press.

Copyright © 2005 by China Machine Press.

本书中文简体字版由美国 Pearson Education 培生教育出版集团授权机械工业出版社独家出版。未经出版者书面许可,不得以任何方式复制或抄袭本书内容。

版权所有,侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号:图字:01-2003-8708

图书在版编目(CIP)数据

嵌入式系统的描述与设计/(美)盖斯基(Gajski D. D.)等著;边计年等译。-北京:机械工业出版社,2005.7

(计算机科学丛书)

书名原文:Specification and Design of Embedded Systems

ISBN 7-111-16422-9

I.嵌… II.①盖… ②边… III.微型计算机-系统设计 IV.TP360.21

中国版本图书馆 CIP 数据核字(2005)第 066897 号

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑:隋 曦

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2005 年 7 月第 1 版第 1 次印刷

787mm×1092mm 1/16·16.75 印张

印数:0 001-4 000 册

定价:33.00 元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换

本社购书热线:(010)68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及收藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T., Stanford, U.C. Berkeley, C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程,而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下,读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑,这些因素使我们的图书有了质量的保证,但我们的目标是尽善尽美,而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正,我们的联系方法如下:

电子邮件: hzedu@hzbook.com

联系电话: (010) 68995264

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037

专家指导委员会

(按姓氏笔画顺序)

尤晋元
石教英
张立昂
邵维忠
周立柱
范明
袁崇义
谢希仁

王珊
吕建
李伟琴
陆丽娜
周克定
郑国梁
高传善
裘宗燕

冯博琴
孙玉芳
李师贤
陆鑫达
周傲英
施伯乐
梅宏
戴葵

史忠植
吴世忠
李建中
陈向群
孟小峰
钟玉琢
程旭

史美林
吴时霖
杨冬青
周伯生
岳丽华
唐世渭
程时端

译者序

多年来,半导体制造技术持续迅速发展,芯片规模的增大使得集成电路的应用范围也在迅速扩大,已经渗透到各行各业,在机械系统和其他系统中广泛使用,形成所谓嵌入式系统。嵌入式系统深入到人类活动的几乎所有方面。从和我们日常生活密切相关的手机、个人数字助理(PDA)、音响设备、高清晰度电视、游戏机,到公共场所常见的汽车电子设备、通信设备、电子医疗设备,再到特殊领域,如军事、航天、水下等使用的电子设备,嵌入式系统无时无刻不在发挥着重要作用并将继续扮演越来越重要的角色。同时,片上系统(SOC)应运而生,在一个芯片上可以同时集成微处理器、存储器、专用电路、输入输出电路、模拟电路,甚至射频电路和微机械。

我们已经有了相当多有关逻辑综合、布局、布线等方面的工具。在目标系统的规模不断增大、功能日益复杂的情况下,传统的设计方法学已经越来越不适应这种变化,迫切需要从更高的层次上重新审视和研究数字系统的设计方法学问题。本书作者提出的设计理念是,在设计早期的更抽象的高层次上开始,进行系统级设计。首先,在高层次上的设计有助于更好地理解系统的功能,保证设计的正确性。其次,在高层次上的决策对整个设计的影响更大,因而,早期的正确决策将得到更大的回报。由于在系统级引入优化策略,可以在各种设计选择中寻求最好的解决方案,因而可以预期基于这种新的方法学会设计出更加有竞争力的产品。

本书是一本很好的介绍嵌入式系统描述和设计技术的教科书。作者根据多年的研究经验和成果,既系统地介绍基本概念,又介绍实际的描述和设计方法。本书几乎论述了嵌入式系统设计的所有方面,包括模型和体系结构、描述语言、系统划分、设计质量评估、描述细化以及系统级方法学等。而且,对于每章涉及的每个问题,作者都予以详细地讲解并给出大量的实例。本书对于初学者和相关领域的从业人员来说,是一本很好的入门书,而对于从事嵌入式系统设计和研究的工程技术人员、科研人员 and 高等院校的本科生和研究生又是一本很详尽的参考书。据我们所知,本书已在全世界很多大学中作为教材或参考书,具有最高的采用率。

本书虽出版于1994年,但其基本的思想和方法仍然是嵌入式系统设计的指导思想和理论基础。本书介绍的描述语言,作者在后来又有所改进,并改称为SpecC。除美国外,还有日本和加拿大等许多国家都在研究和推广这种系统描述语言。

进入2000年以来,在国家的大力支持和推动下,我国在嵌入式系统和片上系统SOC的研究与开发领域出现了前所未有的好势头。我们向读者介绍这本书,相信对我国集成电路工业和技术的发展会发挥一定作用。

本书是许多老师同学共同努力的结晶。吴为民译第1章,刘志鹏译第2章,王海力译第3章,王迪译第4章和附录A,芦琰琰译第5章,童琨译第6、9章,赵康、林锋译第7章,王云峰、赵建洲译第8章。边计年和吴为民分别对所有翻译初稿进行了仔细的审校和修改,最后由边计年统稿定稿。对于有些术语,国内出现不同的说法,我们根据全国科学技术名词审定委员会公布的《计算机科学技术名词》和自己的理解决定取舍。虽然我们的宗旨是准确表达原文的意思,但鉴于译者水平和时间所限,错误和不足在所难免,望读者不吝指正。

译者

2005年1月

英文版序言

基本原理

在过去的十年里,VLSI 的设计技术特别是 CAD 产业取得了很大成功,与 IC 制造技术同步以异常迅猛的速度发展。与高抽象层次相比,低层次的设计问题更早地变得难解和费时。因此,学术界和产业界都被迫将注意力首先投向诸如电路模拟、布局、布线及布图规划等问题。当这些问题变得易于处理时,便成功地开发出用于逻辑模拟和综合的 CAD 工具,并将其引入到设计过程中。随着设计复杂性的大幅度提高和产品进入市场时间要求的大幅度缩短,产业界和学术界都开始关注比逻辑级和布局级更高级别的设计问题。由于高层次的抽象使设计者需要考虑的对象数目降低了一个数量级,也就允许产业界能够在更短的时间内设计和制造出复杂的专用集成电路(ASIC)。

继逻辑综合之后,行为综合也在设计方法学中发挥了提高抽象级别的作用。行为综合是用于设计单个专用集成电路的。当设计方法学要求更高的抽象级别时,这些专用集成电路和标准处理器及内存一起被用做系统的组件。系统级的设计方法学关注系统描述以及从描述到一组互联组件的变换和细化。系统描述依据执行于抽象数据类型上的计算,而变换和细化则包括针对标准处理器进行软件编译和根据定制的组件进行硬件综合。但在这一点上,虽然已有若干年系统制造的历史,尽管有明确的需求,产业界和学术界并未对系统级设计方法学的发展和形成给予充分关注。为解决复杂性问题和缩短设计周期,产业界最近已开始关注于建立一个一致性的系统级设计方法学。

强调更抽象化和系统级方法学的主要原因是基于这样一个事实:高层次抽象更接近设计者的日常思维习惯。比如,难以想像设计者仅靠电路原理图如何能够对一个由 10 万个门或 10 万个布尔表达式组成的系统设计进行描述、建立文档及进行交流。系统越复杂,当设计者用电路级、逻辑级、寄存器级原理图对其进行描述时,就越会产生功能理解上的困难。另一方面,当系统被描述成操作于抽象数据类型上的一系列复杂计算,并且通过抽象通道进行通信时,设计者将发现这更易于描述和验证特定的功能,更易于采用不同技术对各种实现进行评估。

必须承认,对系统级设计的研究虽已有很多年,但目前仍在相当程度上局限于特定领域和团体。例如,计算机体系结构界考虑的是将计算和算法映射到不同体系结构的方法。体系结构可以是脉动阵列、超立方体、多处理器以及大规模并行处理器。软件工程界则一直在研究软件代码的描述和设计方法。CAD 界集中于接口综合、内存管理、系统描述捕捉以及设计空间探索等系统问题。但是,很多问题仍悬而未决,其中最重要的是缺乏一个得到普遍接受的理论框架和支持系统设计方法学的 CAD 环境。尽管有这些悬而未决的问题,系统设计技术也已成熟到一定程度。因此,用一本书对发展至今的基本概念和成果进行总结,希望会对系统设计领域的学生和从业者有所帮助。在本书中,我们力图涵盖多种研究项目的思想和成果。由于这个领域还比较年轻,我们仍可能忽略了某些很有趣和有益的项目,为此我们在此表示歉意,并希望得到这些技术的信息,以便将它们结合到将来的版本中。同样,由于各种原因,我们也没

能在本书中详细地论及系统级的若干重要问题,包括形式验证、测试设计以及协同模拟。尽管如此,我们相信,一本有关系统级描述和设计的书将有助于电子系统设计自动化(ESDA)在未来的发展和繁荣。

读者

本书面向计算机科学和工程界的三类读者。首先,它会吸引系统设计者和工程管理者,因为他们会对 ASIC 和系统设计方法学、软硬件协同设计以及设计过程管理感兴趣。其次,该书也可以被 CAD 工具的开发者的所利用,他们会采用该书中所给出的现存或未来工具中的一些概念进行系统描述捕捉、设计空间探索以及系统建模和细化。最后,由于该书综述了系统设计的基本概念,提出了包括软件和硬件在内的系统设计方法学的原理,因此,对于一门高年级本科或研究生课程来说,该书也是有价值的。该课程可针对有志于计算机体系结构、设计自动化和/或软件工程的学生。

本书的组织

本书组织成 9 章,分为四个部分。第 1、2 章提出系统设计的一些基本问题,讨论了可用于系统行为捕获和实现的各种概念模型。第 3、4、5 章论及用于描述系统功能性的语言,还涉及通过模拟验证系统功能性的各种问题。第 6、7、8 章提供了一个有关系统划分、评估和模型细化的算法和技术的综述。第 9 章将所有这些主题结合成一个一致性的设计方法学,包括对系统设计通用环境的讨论。

在对第 1、2 章所定义概念理解的前提下,每一章都是独立的,可以单独阅读。本书每一章都采用了相同的写作风格和组织方法。典型的一章包含了介绍性的例子,定义了基本概念,阐述了要解决的主要问题。针对所提出的问题,每一章还描述了若干著名的算法或解法,并解释每种方法的优缺点。每一章还包括对该领域其他工作的小综述,以及对发展方向的讨论。

在每章的末尾,我们还给出若干练习,分为三类:家庭作业类问题,项目类问题,以及课题类问题。家庭作业类问题旨在检测读者对每章基本内容的理解。项目类问题用一个星标记。为解决项目类问题,读者需要在对一些文献进行研究的基础上才能对题目有更深入的理解。这类问题可能需要学生用若干星期时间来完成。课题类问题用两个星标记,是尚未解决的问题,若深入研究,将是不错的硕士或博士学位论文课题。

本书可用于两种不同的课程。一种可侧重于系统描述、建立文档以及验证方面,跳过第 6、7、8 章中的算法部分。另一种课程,侧重设计方法学和设计空间探索技术,跳过语言和模拟方面的内容。无论采用哪种课程,我们认为本书都会有助于填补计算机科学与工程课程中的一个空白,即除了涵盖电路设计、逻辑设计以及计算机体系结构的内容,还讲述了系统设计技术。

我们希望本书的内容选择和写作风格接近您的期望。我们欢迎您的意见和建议。

Daniel Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong
加州大学艾尔温分校

致 谢

在过去六年中,我们同许多同事和学生讨论过系统设计的基本问题。感谢他们,因为没有这些讨论,很多问题就不会得到澄清,很多思想也不会得到尝试。

感谢帮助我们形成本书的选题和确定写作材料的人们。特别要感谢 SRC 的 Peter Verhofstadt 和 NSF 的 Bob Grafton 鼓励我们在此领域进行研究,以及 Rockwell 国际公司的 Bob Larsen 多年来帮助我们开发质量度量方法和用于比较分析的客观方法。

我们对加州大学艾尔温分校的 Nikil Dutt 教授、Fadi Kurdahi 教授和 Queensland 大学的 Sri Parameswaran 教授致以谢意。在几次会议期间,他们提供了有价值的意见,指出了系统设计的相关工作。我们也感谢松下研究和开发实验室的 Peter Fung 为我们的设计方法学所做的证实工作。我们同时感谢加州大学艾尔温分校 CAD 实验室的以下人员:Smita Bakshi、Viraphol Chaiyakul、Tedd Hadley、Nancy Holmes、Pradip Jha、Erica Juan、Raghava Kondepudy 以及 Loganath Ramachandran,他们的审校工作和提出的有价值的建议使本书更易于理解。感谢 Sarah Wilde 和 Judy Olson 的编辑工作和在理解复杂技术材料时所付出的耐心。我们对 Jon Kleinsmith 在协调作者和审稿者过程中所做的工作表示感谢。

该项工作受到美国自然科学基金(批准号 MIP-8922851)和半导体研究公司(批准号 93-DJ-146)的部分资助。作者对他们的支持表示感谢。

目 录

出版者的话	
专家指导委员会	
译者序	
英文版序言	
致谢	
第1章 引言	1
1.1 设计表示	1
1.2 抽象级别	2
1.3 当前的设计方法学	3
1.4 系统级方法学	5
1.5 系统描述和设计	6
第2章 模型与体系结构	9
2.1 引言	9
2.2 模型分类	11
2.3 面向状态的模型	11
2.3.1 有限状态机	11
2.3.2 Petri 网	13
2.3.3 层次化并发有限状态机	15
2.4 面向活动的模型	16
2.4.1 数据流图	16
2.4.2 流程图	17
2.5 面向结构的模型	18
2.6 面向数据的模型	18
2.6.1 实体-关系图	18
2.6.2 Jackson 图	19
2.7 异构模型	20
2.7.1 控制/数据流图	20
2.7.2 结构图	20
2.7.3 程序设计语言模式	22
2.7.4 面向对象的模型	23
2.7.5 程序状态机	24
2.7.6 队列模型	25
2.8 体系结构分类	26
2.9 专用体系结构	26
2.9.1 控制器体系结构	26
2.9.2 数据通路体系结构	26
2.9.3 带数据通路的有限状态机	28
2.10 处理器	28
2.10.1 复杂指令集计算机	28
2.10.2 精简指令集计算机	29
2.10.3 向量机	30
2.10.4 超长指令字计算机	31
2.11 并行处理器	31
2.12 结论	33
2.13 练习	33
第3章 系统描述语言	35
3.1 引言	35
3.2 概念模型的特性	36
3.2.1 并发性	36
3.2.2 状态迁移	38
3.2.3 层次化	39
3.2.4 程序结构	40
3.2.5 行为完成	41
3.2.6 通信	42
3.2.7 同步	43
3.2.8 异常处理	45
3.2.9 非确定性	46
3.2.10 时序	46
3.3 嵌入式系统的描述要求	47
3.4 描述语言综述	48
3.4.1 VHDL	48
3.4.2 Verilog	50
3.4.3 HardwareC	51
3.4.4 CSP	52
3.4.5 Statecharts	53
3.4.6 Argos	55
3.4.7 SDL	55
3.4.8 Silage	56

3.4.9 Esterel	56	5.4.2 分叉	88
3.5 SpecCharts	57	5.5 异常处理	89
3.5.1 语言描述	57	5.6 从程序状态机到任务	90
3.5.2 用 SpecCharts 描述嵌入式系统	59	5.6.1 概述	90
3.5.3 等价图图形化表示	61	5.6.2 算法	91
3.5.4 语言的可扩展性	61	5.6.3 时间调整	92
3.6 结论和发展方向	62	5.6.4 综合	93
3.7 练习	63	5.7 结论和发展方向	93
第 4 章 系统描述举例	65	5.8 练习	93
4.1 引言	65	第 6 章 系统划分	95
4.2 电话应答机	65	6.1 引言	95
4.3 用 SpecCharts 进行系统描述	67	6.2 结构划分和功能划分	95
4.4 测试用例举例	72	6.2.1 结构划分	95
4.5 可执行系统描述的优点	73	6.2.2 功能划分	96
4.6 PSM 模型的优势	74	6.3 划分中的问题	97
4.6.1 层次性	74	6.3.1 系统描述抽象级别	98
4.6.2 状态迁移	74	6.3.2 粒度	98
4.6.3 程序设计结构	74	6.3.3 系统组件的分配	99
4.6.4 并行性	75	6.3.4 度量和评估	99
4.6.5 异常处理	75	6.3.5 目标函数和接近函数	100
4.6.6 完成	75	6.3.6 划分算法	101
4.6.7 状态分解和编码的等价性	75	6.3.7 输出	101
4.7 实验	75	6.3.8 控制流程和设计者的参与	101
4.7.1 系统描述的获取比较	75	6.3.9 典型系统配置	102
4.7.2 系统描述的理解比较	76	6.4 基本划分算法	102
4.7.3 系统描述的量化比较	76	6.4.1 随机映射	102
4.7.4 设计质量比较	77	6.4.2 层次化结群	102
4.8 结论	78	6.4.3 多级结群	104
4.9 练习	78	6.4.4 成组移动	105
第 5 章 转换成 VHDL	81	6.4.5 比率切割	107
5.1 引言	81	6.4.6 模拟退火	108
5.2 状态迁移	82	6.4.7 遗传进化	109
5.3 消息传递通信	84	6.4.8 整数线性规划	110
5.3.1 阻塞式消息传递	84	6.5 硬件功能划分	110
5.3.2 非阻塞式消息传递	86	6.5.1 Yorktown 硅编译器	111
5.4 并发	86	6.5.2 BUD	113
5.4.1 数据流	87	6.5.3 Aparty	115

6.5.4 其他技术	117	7.5.2 Aparty	162
6.6 软硬件划分算法	118	7.5.3 Vulcan	162
6.6.1 贪心算法	118	7.5.4 SpecSyn	162
6.6.2 爬山算法	119	7.6 结论和发展方向	164
6.6.3 二分约束搜索算法	120	7.7 练习	164
6.7 系统功能划分	120	第8章 设计描述细化	167
6.7.1 Vulcan	120	8.1 引言	167
6.7.2 Cosyma	121	8.2 细化变量群组	167
6.7.3 SpecSyn	122	8.2.1 变量折叠	167
6.7.4 其他技术	123	8.2.2 存储地址转换	168
6.8 折中的探索	124	8.3 通道细化	169
6.9 结论和发展方向	125	8.3.1 通道和总线的表征	169
6.10 练习	125	8.3.2 问题的定义	170
第7章 设计质量评估	127	8.3.3 总线生成	170
7.1 引言	127	8.3.4 协议生成	176
7.1.1 精确性与速度	128	8.4 解决访问冲突	178
7.1.2 评估的保真度	128	8.4.1 仲裁模型	178
7.2 质量度量	129	8.4.2 仲裁方案	179
7.2.1 硬件成本度量	129	8.4.3 仲裁器生成	180
7.2.2 软件成本度量	130	8.5 细化不兼容接口	181
7.2.3 性能度量	130	8.5.1 问题的定义	182
7.2.4 其他度量	134	8.5.2 通信协议描述	182
7.3 硬件评估	135	8.5.3 接口进程生成	184
7.3.1 硬件评估模型	135	8.5.4 协议兼容的其他方法	189
7.3.2 时钟周期评估	136	8.6 细化软件/硬件接口	191
7.3.3 控制步评估	141	8.6.1 目标体系结构	192
7.3.4 执行时间评估	145	8.6.2 变量分配	192
7.3.5 通信速率评估	147	8.6.3 接口生成	194
7.3.6 面积评估	148	8.6.4 数据访问细化	195
7.3.7 引脚评估	156	8.6.5 控制访问细化	197
7.4 软件评估	156	8.7 结论和发展方向	198
7.4.1 软件评估模型	156	8.8 练习	199
7.4.2 程序执行时间	159	第9章 系统设计方法学	201
7.4.3 程序存储大小	160	9.1 引言	201
7.4.4 数据存储大小	160	9.2 基本概念	201
7.5 系统级工具的评估技术	161	9.3 设计方法学举例	201
7.5.1 BUD	161	9.3.1 当前的惯例	204

9.3.2 系统级方法学	205	9.5 系统设计的概念化环境	213
9.4 通用综合系统	206	9.6 结论和发展方向	215
9.4.1 系统综合	208	9.7 练习	216
9.4.2 ASIC 综合	209	附录 A 应答机的自然语言描述	217
9.4.3 逻辑综合和时序综合	211	附录 B 应答机的 SpecCharts 描述	219
9.4.4 物理设计	212	参考文献	233
9.4.5 软件综合	212	术语解释	247
9.4.6 系统数据库	213	索引	249

第1章 引言

在过去的20年里,设计一个系统所需经过的实际步骤没有实质性的变化。另一方面,各设计步骤的侧重点却有了显著变化:由于较后的阶段已或多或少地实现了自动化,设计者越来越关注于系统设计过程较早期的更抽象阶段。这种关注点的转换使设计者能够在更短的时间内设计出愈加复杂的系统。设计这样的复杂系统,达到功能正确性要比达到芯片面积或程序所占内存最小化要重要且复杂得多。系统的功能性在早期的设计步骤中能得到最好的理解,此时很多的实现细节还没有被加入。这就是为什么在系统设计过程中早期的阶段至关重要。

在本章中,我们将从多种角度着重探究这种关注点的转变。我们将描述系统设计的相关步骤、设计表示的分类、各步骤的抽象层次,给出曾有的关注点及其如何转变的历史,描述下一次可能会向系统级的转变,以及在系统级所需的工具和方法学。

1

1.1 设计表示

对任何特定的产品,设计过程通常开始于对产品功能的概念化,结束于对产品制造蓝图的制订。在设计完成之前,很多不同的人员会参与其中。

例如,需要市场部门研究市场需求和确定对新产品的要求。需要一个首席设计师将这些要求转换为产品的体系结构。技术专家介入选择采用的工艺、可能需要的组件和供应者,而负责计算机辅助设计和计算机辅助软件工程的小组必须获得或开发出支持产品各部分设计的工具。负责设计的小组将制订出蓝图,指明如何从可获得的并采用了所选工艺的组件来制造出产品。软件工程师将为产品所采用的处理器编写代码。测试工程师需要制定测试策略和测试向量以确定产品的可靠性,而制造工程师则需要为产品的实际制造定义机器操作和制定工厂生产进度表。

每个小组都是从各自的角度看待产品,需要特定的信息支持其特定的工作。这样,每个产品、进而每个设计,必然有若干个不同的表示或视角,侧重于不同类型的信息。对于单一的表示也是这样,这时随着设计的周期进展可获得不同层次的细节。

3种最常采用的表示法分别针对产品的行为、结构和物理方面。

行为表示 将设计简单地看做黑盒,并将其行为描述为输入值和终止时间的函数。换句话说,行为表示描述系统的功能性,但不涉及任何实现。行为表示定义了黑盒如何对输入值的任何组合做出响应,但不提供关于如何设计这个黑盒的任何指示。

2

结构表示 与行为表示比较而言,开始解答一些设计问题,因为结构表示将黑盒定义成一组组件及其互联关系。换句话说,结构表示着重于描述产品的实现。虽然黑盒的功能性可从互联的组件中得出,但并不明确描述功能性。

物理表示 承载了设计的更进一步的实现,以结构化表示描述了组件的特点。例如,物理表示会提供每个组件的尺寸和位置,以及组件之间互联的物理特性。这样,结构表示提供了设计的互联性,而物理表示则描述了互联组件之间的空间关系,描述了被制造设计系统的重量、大小、散热、功耗以及每个输入输出引脚的位置。

一般来说,设计系统的过程应遵循从行为表示到结构表示再到物理表示的路线,并沿此路

线逐渐获得实现细节。如上所述,虽然我们需要这些实现细节来制造产品,但这些细节也会引起系统功能性的模糊,阻碍了设计者确保系统正常工作的努力。举一个例子:考虑一个能做 32 位数字加减的简单系统。行为表示仅包含两个等式: $a := a + b$ 和 $a := a \times b$ 。而结构表示则可能包含若干互联的寄存器、算术单元以及多路选择器,使得系统功能性难以辨别,尤其组件数量很多或者某组件的功能仅被部分使用时更是如此。因此,如果要强调功能正确性这个正在不断变成至关重要的问题,我们就应该承认:当工作于行为表示而不是工作于结构或物理表示时,设计者将会更成功从而设计出更好的产品。

3

1.2 抽象级别

在前一部分中,我们描述了最常见的几种设计表示类型,即行为、结构和物理表示。到此,我们可以进入下一个阶段,并注意到在电子系统的设计中,每种表示类型适合于若干不同的抽象级别或者粒度。不同的抽象级别基于它们所使用的对象类型进行区分,归入四类:晶体管级、门级、寄存器级和处理器级。在图 1-1 中总结了这些不同的抽象级别与各种表示类型的关系。

级 别	行 为 表 示	结 构 组 件	物 理 对 象
晶体管	微分方程,电流-电压关系图	晶体管,电阻,电容	模拟和数字单元
门	布尔方程,有限状态机	门,触发器	模块,元件
寄存器	算法,流程图,指令集,广义有限状态机	加法器,比较器,寄存器,计数器,寄存器文件,队列	微芯片,专用集成电路
处理器	可执行描述程序	处理器,控制器,存储器,专用集成电路	印刷电路板,多芯片模块

图 1-1 设计表示和抽象级别

根据图 1-1,在晶体管级的主要组件是晶体管、电阻和电容。这些对象可组合起来,形成满足给定功能的模拟电路和数字电路。在该级别上,功能通常用一组微分方程或者某类电流-电压关系描述。最后,这样一种电路的物理表示,称为单元,将包含晶体管级元件和它们之间的连线。这种单元通常根据它们所包含元件的版图来定义。

4

在门级,主要的元件是逻辑门和触发器。逻辑门是用于执行布尔运算的特殊电路,如“或”和“与”。触发器是基本存储单元,每个仅能存储一位信息。这些门和触发器表示典型的数字单元。可将这些独立组件结合起来,安置在硅片表面,以构成算术和存储模块,其行为可通过逻辑方程和有限状态机图描述。

在寄存器级的主要元件是由门和触发器,如加法器、比较器、多路选择器、计数器、寄存器、寄存器文件、数据缓冲器和队列设计而成的算术和存储单元。每个寄存器级元件是一个物理对象,具有固定尺寸、固定传播时间,并且在模块边界上其输入和输出有固定位置。寄存器级元件可用于微芯片设计,该设计可用流程图、指令集、广义有限状态机或状态表描述。

最后,最高抽象级别称为处理器级,因为该级别的基本组件是处理器、存储器、控制器、接口,以及称为专用集成电路(application-specific integrated circuit, ASIC)的定制微芯片。一个或多个这样的组件可安置于一个印刷电路板(printed-circuit board, PCB)上,微芯片焊接于其上,并通过印制于板上的导线连接起来。为降低板的尺寸,可用硅基片代替 PCB 用于微芯片

的连接,这种情况下的封装称为多芯片模块(multi-chip module, MCM)。由处理器级组件组合而成的系统可用若干种不同的方式描述其行为:如采用自然语言,采用以硬件描述语言方式的可执行描述,或者采用算法或程序设计语言程序。

现在,重要的是记住这样一个事实,设计者只能将他们的努力关注于他们所能理解的系统级别上,这个级别很大程度上决定于需要确保对象数量相对小。例如,一个设计者可能理解一个包含 10 个布尔方程的系统,但肯定不能理解一个包含 1 万个方程的系统。在后一种情况下,他将不得不向高抽象级别转移直至达到系统能表示成可管理的对象数量,如表示成 10 个算法。在较低抽象层次上,只有当我们将系统划分为小块并在许多设计者中分配时,或者当我们采用自动工具建造它时,系统才是可管理的。幸运的是,随着新的在较低层次上的设计工具出现,设计者能自由地关注于高级别,而在高级别做出决策要比在较低级别对质量的影响大得多。

每个电子系统在经历从概念化到制造的设计过程中,不可避免地要经过大部分这些抽象级别。在设计过程中的这组特定任务,它们执行的特定顺序和每个任务执行过程中要使用的 CAD 工具,称为设计方法学。在下一部分中,我们将简要讨论过去和现在在工业环境中曾占支配地位的设计方法学。

1.3 当前的设计方法学

在过去的 25 年中,大多数 ASIC 和系统生产厂家采用一种基于“捕获-模拟”(capture-and-simulate)的设计方法学。这种设计方法学开始于对产品的一组特定需求,该需求通常由市场部门提供。由于这些需求不能囊括有关产品实现的任何信息,故由首席设计师组成的小组提出一个芯片体系结构的粗略结构图,作为虽不完备但初步的系统描述。在某些情况下,这个初步结构图在提交给逻辑和版图设计者小组之前还要被进一步细化,而逻辑和版图设计者的任务是将每个功能模块转换为逻辑或电路原理图(最终被原理图捕获工具所捕获),通过模拟方法对功能、时延及故障覆盖率进行验证。这个被捕获的原理图,在门阵列技术中可用于驱动物理设计工具进行门的布局和布线,或在定制技术中用于在布局和布线前将门映射到标准和定制单元。

仅仅在最近几年,逻辑综合才被公认为是设计过程构成整体所必需的部分,而这种认可导致了在设计方法学上的革命性变化,因为“捕获-模拟”方法正逐步让位于“描述-综合”(describe-and-synthesize)方法学。这个新方法学的优势是它允许我们用纯行为形式描述一个设计,而不包括实现细节。特别地,我们能采用布尔方程和有限状态机图描述设计。在这个方法学中,设计结构采用 CAD 工具自动综合生成,而不是用手工综合而成,因为除非对于简单电路,手工综合是一项非常乏味的工作。

“描述-综合”方法学可应用于若干抽象层次。在门级,功能和控制单元可采用逻辑综合(logic synthesis)方法进行综合。例如,功能单元如 ALU、比较器及多路选择器等,可用布尔方程描述,然后通过两个阶段进行综合。第一个阶段,称为逻辑最小化(logic minimization),使布尔方程中“与”和“或”运算符的数量(或者在相等意义上,文字的数量)达到最小化,同时满足代价和时间约束。第 2 个阶段,称为工艺映射(technology mapping),将这些已最小化的布尔方程采用逻辑门实现,这些逻辑门来自于选定工艺技术的门库。

另一方面,控制单元采用有限状态机图定义,然后通过两个阶段综合。第 1 个阶段,称为

状态最小化(state minimization),使状态数达到最小化,并为每个状态分配2进制编码,使得实现状态和输出函数所需代价降低。在第2个阶段,由布尔方程定义的次态和输出函数通过逻辑最小化和技术映射进行优化,方法如前所述。

在寄存器级,表示处理器、内存及ASIC的微芯片,可采用行为综合(behavioral synthesis)或高层次综合(high-level synthesis)技术进行综合。这些微芯片的结构包含功能、存储及控制单元,这些单元预先已设计好并存储于寄存器级库中。这些微芯片的行为可通过程序、算法、流程图、数据流图、指令集的方式描述,或通过广义有限状态机描述,在其中的每个状态可执行任意复杂的计算。

7

我们通过完成3个主要的综合任务将行为描述转换成结构描述:分配、调度、绑定。分配(allocation)的任务是确定用于实现微芯片所需的寄存器级组件或资源的数量。换句话说,确定功能单元的数量,每个功能单元执行的操作,流水线级数,每个操作的时延,以及每个单元的代价和大小。分配还必须确定存储单元的数量,如所需的寄存器、寄存器文件、队列及内存。除了每个单元的大小、代价以及所需端口数之外,还必须确定每个存储单元的读出和写入所需访问时间。最后,分配还必须确定系统中每种总线的数量、大小、协议及时延,以及为连接功能单元和存储单元到这些总线所需的各种操作。在很多种情况下,分配提供了一个在代价-性能折中方面进行探索的机会,这是非常重要的,因为进行更大量的对诸如功能、存储、互联单元等资源的分配,可提高性能,但一般也增加了代价。

一旦资源被分配,调度(scheduling)的任务就是将行为描述划分成时间间隔,称为控制步。在每个控制步期间,通常是一个时钟周期长度,数据从一个寄存器传输到另一个寄存器,且若有必要,还在传输过程中由功能单元进行变换。在每个控制步,所有的寄存器传输都是并发执行的。这样,设计的性能大致与在每个控制步中可得到的资源数量成比例。

必须注意,虽然调度任务可确定在每个控制步中的所有操作,但却不将操作分配到特定的寄存器级组件中。这项工作由绑定(binding)任务来完成,绑定的任务是将变量分配到存储单元,操作分配到功能单元,并且对于每个从存储单元到功能单元及反方向的数据传输,确保能分配一条特定的通信路径或总线。

8

通过采用逻辑和行为综合,“描述-综合”方法学能使设计者用不含任何时延、工程或工艺信息的行为描述方法来描述微芯片的功能,然后自动综合成包含寄存器组件的结构描述,并随后综合成门级元件。

逻辑和行为综合的意义是,在过去的十年中,它们已成功地将“捕获-模拟”方法学转换为“描述-综合”方法学。逻辑综合较早为设计界所接受,因为设计者发现,逻辑综合容易吸收进“捕获-模拟”方法学,其原因是这种方法学已经关注于捕获和模拟门级原理图或网表,故已有必需的图形捕获工具、模拟器、库和框架。换句话说,显然逻辑综合能通过优化其捕获的原理图,给“捕获-模拟”方法学附加额外的价值。一旦逻辑综合被接受,设计者就开始采用布尔表达式来描述逻辑,取代用图形捕获工具来捕获门的方法。最终,这种新技术鼓励了通过行为描述而不是原理图来捕获设计的实践,尤其是采用行为描述后,导致了生产率的大幅度提高。

行为综合比逻辑综合花费较长时间被接受的主要原因是缺乏对行为综合提供支持的基础。换句话说,没有在寄存器级的图形捕获工具,设计者完全不习惯通过硬件描述语言如VHDL来捕获设计。最基本的问题是,设计者受到的是以结构而不是以行为来进行思维的训练,而新方法要求他们应对若干新的问题。比如,第1个问题是,每个设计可用若干种方式描

述,采用不同的硬件描述语言结构。由于描述的风格对综合后设计的质量有重大的影响,行为综合就要求对综合算法和 CAD 工具的工作有深入理解。第 2 个问题,抽象的寄存器级不带有可模拟模型的组件库,因此每个设计小组不得不开发自己的模型。最后,直至最近才出现探索工具环境,这种工具可供设计者做关键性的决策,甚至在使用行为综合工具对剩余部分进行设计之前,部分地描述设计。正是在处理这些问题的过程中,设计者逐渐发现:关注于对行为的正确描述,然后使用自动工具对大量的可能解进行探索和比较,会产生巨大的生产效益。

9

现在,在成功引入逻辑和行为综合工具后,系统设计者和 CAD 业界一致的疑问是:“描述-综合”方法学能否扩展,应用于整个系统,包括软件和硬件设计。换句话说,如果继续向这个趋势发展到芯片级以上,关注于更高的抽象级别,则生产率可能会得到更大的提高。在下一部分,我们将对这种系统级方法学的需求和实质问题加以讨论。

1.4 系统级方法学

一般而言,在较高抽象级别上的设计方法学尚未很好地确立。如前所述,市场部门通常定义市场需求,而首席设计师给出非正式的框图,在进行某些初始设计后,从中可得出设计描述。产生框图的那些设计决策,通常是基于特定设计师的个人经验,而不是基于对所有可能的结构和工艺选择的探索。此外,这些框图通常是在缺乏对系统功能性全面了解的情况下产生的。这种对系统功能性定义的滞后经常导致更长的设计周期,因为在设计后期发现的矛盾需要很耗时的设计反复。

与这种以特定的方式获得功能性相反,更可取的方法是,在设计的最早期阶段,在尚未做出任何设计决策之前,将更多的努力投入到对系统功能的描述上,因为这种早期的努力会获得巨大而全面的回报。特别地,与设计描述,尤其是与可执行的设计描述打交道有很大的优点,因为这不仅能捕获系统的功能性,而且还能被市场部门利用来研究产品在市场上的竞争能力。另外,可执行的系统描述在设计过程的所有步骤中均可作为文档,尤其是在促进并发工程方面,能够对分派给设计小组中不同成员的不同子系统清晰地定义功能性和接口。而且,这些子系统中的一个发生变化都易于处理,其对系统其他部分的影响可快速估计。可执行的系统描述还适合于对系统的不同设计性质及功能性进行自动验证。此外,可执行的系统描述的优势还在于不仅允许功能验证的自动化,而且一旦做出了适当的工艺决策,还允许设计探索和设计综合的自动化。最后,可执行的系统描述还能持续作为所有产品在其生命期中进行升级的起点,并支持产品维护。

10

一旦我们采用了可执行的系统描述,则选择用来编写这些描述的语言就成为系统设计方法学中最主要的问题之一。为了和 CAD 工具接口,这样的语言必须易于捕获、理解和使用。这种语言必须能够捕获系统的所有特征,并易于对实现进行综合。最后,这样的语言应能以易读的和完备的而又不是过度复杂的方式对系统及其实现进行建模。

语言选择并不是在定义系统设计方法学中遇到的唯一问题。一旦系统被描述后,确保所选择的系统方法学能允许在设计选择间容易地进行探索,也是必要的。在这样的方法学中,首先必须允许设计者分派结构组件和约束。结构组件就是处理器、存储器及 ASIC。处理器由其指令集和每个指令的执行速度来定义;存储器由其大小、读/写协议及存取时间来定义;ASIC 由其门数(门阵列)或晶体管数(定制设计)表示的大小、门或晶体管的传播时延、封装尺寸及所允许的功耗等来定义。

11 接下来,设计描述必须被划分成软件部分和硬件部分。软件部分将用软件实现并在一个或多个指定的处理器上执行,而硬件部分将综合成一个或多个 ASIC。系统的软件部分可进一步划分成两个或多个部分,每个部分在独立的处理器上运行。不难想像一个主处理器运行较慢的系统功能同时又有有一个或多个协处理器执行快速数据转换的情况。类似地,硬件部分可能也不适合用单一的 ASIC 实现,即可能需要用若干个 ASIC 实现。

由于对每个不同的处理器组件的指派和不同的划分将会产生不同的系统实现,若对各种选择进行评估,就需要设计者对每种实现的诸如性能、成本、功耗、测试和封装成本等质量度量进行估计。将估计出的每组质量度量与给定的要求进行比较,并选择其中对需求满足最好的实现。这样,这种得自于设计描述的设计探索将能使设计者寻找到最有成本效益的解决方案。

一旦寻找到最佳解决方案,就需要对设计描述进行细化以反映指派和划分方面的决策,使得设计描述的不同部分被移到适合的组件中,并在独立的部分中保持通信联系。

12 细化完成后,设计描述将反映出产品的体系结构,正像首席设计师设计的框图所反映的一样。系统组件及其相互通信都得到明确定义,每个组件有其自己的描述,并可在计算被分配到处理器时,由标准的编译器进行编译;或者在计算被分配到定制的 ASIC 时,由行为和逻辑综合工具进行综合。当然,在细化的系统描述和设计师的框图之间会存在差别:首先,细化的系统描述是在对大量解决方案进行彻底和有组织的探索后获得的;其次,细化的系统描述是从原始系统描述中形式化地得出的,因而更具一致性,消除了高代价的、耗时的设计反复。

1.5 系统描述和设计

在前面部分中,我们已简要地描述了基于“描述-探索-细化”模式的系统级设计方法学。这种方法学的优势预示着生产率的大幅度提高,因为精确的系统描述、自动探索及细化等技术能帮助我们避免由多次设计反复而造成设计周期的大幅度增加。这种方法学只需要设计者选择工艺、分派组件和描述需求,然后在一天内完成对数百个可选择设计的自动探索。最后,当做出结构和工艺决策时,加入更多结构上的细节以细化系统描述。重复进行该过程直至达到一个完整的结构描述,包含组件库所定义的适当抽象级别的组件。

我们相信,为了适应不断增加的系统复杂性和不断缩短的上市需求,必须确立一种清晰和有效的系统设计方法学,并将其引入到设计过程中。为此,本书综述各种有关嵌入式系统的系统描述、探索及细化的技术。我们首先简要综述用于软件和硬件系统的捕获、分析及实现的不同模型(第2章)。接着我们阐述嵌入式系统的基本特点,还对若干种系统描述语言进行了综述,并在对这些基本特点支持优劣的基点上,对这些系统描述语言进行了比较(第3章)。为了示范系统描述的技巧,我们针对一个小型的嵌入式系统——电话应答机,设计了一个完整的可执行系统描述。我们还示范了对系统描述语言的适当选择,如何能够简化编写、理解及细化系统描述(第4章)。为了证实可执行系统描述的正确性,我们需要对其进行验证,或者以预先定义的测试向量对其进行模拟。对于后一种情况,该系统描述需要转换成某种模拟器能接受的标准模拟语言。为此,第5章综述了各种转换技术,并以 IEEE 标准的 VHDL 语言为例,对这些技术进行了示范。

13 有关系统探索的过程的内容涵盖于对系统描述进行划分和设计质量度量进行评估的讨论中。在这些领域中,我们综述了有关对系统描述进行划分的各种算法和技术(第6章),以及用于对质量度量如软件性能、数据和程序所占存储器大小、硬件性能、时钟周期、微芯片面积以及

封装代价等进行评估的技术(第7章)。在对系统彻底地进行了探索之后,需要对系统描述进行细化以反映在探索阶段所做出的决策。为此目的,综述了用于对系统描述进行细化的各种算法和技术(第8章)。在最后一章(第9章),将前八章的内容进行综合,形成一个整体一致的系统级设计方法学,并说明了其在整个设计过程中所扮演的角色。

我们相信,本书的内容将揭示出系统设计的艺术,帮助读者为他们的应用和环境选择最好的系统设计方法学。

第2章 模型与体系结构

设计一个系统的第一步应详细指定其功能性,而系统描述的第一步应该准确勾勒出其功能性的具体含义。可以利用多种概念模型来帮助我们以一种系统化的方式理解和组织系统功能。本章纵观几种在硬件和软件系统中最为常用的概念模型,以及实现这些系统所使用的多种体系结构。

2.1 引言

系统设计是指用一组物理组件实现所期望的功能性的过程。很明显,系统设计的整个过程必须从详细指定系统期望功能性开始。但是,这并不是一件容易的工作。作为一个例子,我们来考虑描述一个电梯控制器的任务。如何尽可能详细地描述这样的功能:在任意按键顺序之后完全准确地估计出电梯的位置?用自然语言进行系统描述的问题在于它们是有歧义且不完全的,缺乏提供给这样一个系统所需要的细节的能力。因此,我们需要一种更为精确的方法来详细指定功能性。

为达到我们所要求的准确度级别,最为普遍的方法是把系统想像成由较为简单的子系统或者小模块所组成的集合。下一节将要介绍,至少有5种方法可以把系统功能分解成较为简单的小模块。基本上,区分这些方法就要看分成的小模块的类型以及把这些小模块组合起来构建系统功能性时所遵从的规则。我们把每一种特别的方法称为一个模型(model)。

为了保证可用性,模型必须具备某些特性。首先,模型应该是形式化的,使其不存在歧义。同时它也应该是完整的,这样才能描述整个系统。此外,对于设计者来说他所用的模型应该是易于理解的,也应该是易于修改的,因为设计者在某些点上需要对系统的功能性修改是不可避免的。最后,模型应该足够自然的,而对设计者对系统的理解起到帮助作用而不是阻碍作用。

需要注意的是,一个模型是由对象和组合规则所组成的形式化系统,用于对系统特性的描述。通常情况下,我们使用一个特定的模型把系统分解成小模块,然后再通过使用特定的语言对这些小模块进行描述,从而生成系统描述。一种语言可以描述多种不同的模型,而一种模型也可用多种不同的语言描述。

使用模型的目的是要给出系统的抽象视图。比如说,图2-1中给出了电梯控制器的两种不同模型,图2-1a是它们的自然语言描述。这两种模型之间的区别在于:图2-1b使用一组程序语句来表示控制器,而图2-1c使用状态机来表示控制器。

正如我们所看到的,每一个模型都表示一组对象以及这些对象之间的相互关系。举例来说,状态机模型是由一组状态以及这些状态之间的迁移所构成的;相对地,算法模型是由分支和循环的控制顺序下所执行的一组语句所构成的。允许这些不同的模型在设计中出现的好处就在于它们可以使设计者表示系统的不同视图,从而表示出系统的不同特性。例如,状态机模型非常适合表现一个系统的时态行为,因为它允许设计者显式地表示状态和由于外部或内部事件导致的状态转移。而另一方面,算法模型则没有外在的状态表示。但是,因为它可以利用一连串的语句详细指明系统的输入-输出关系,所以它非常适合于表示系统的过程视图。

设计者在设计过程的不同阶段会选取不同的模型,为的是在特定的时间内强调那些令他

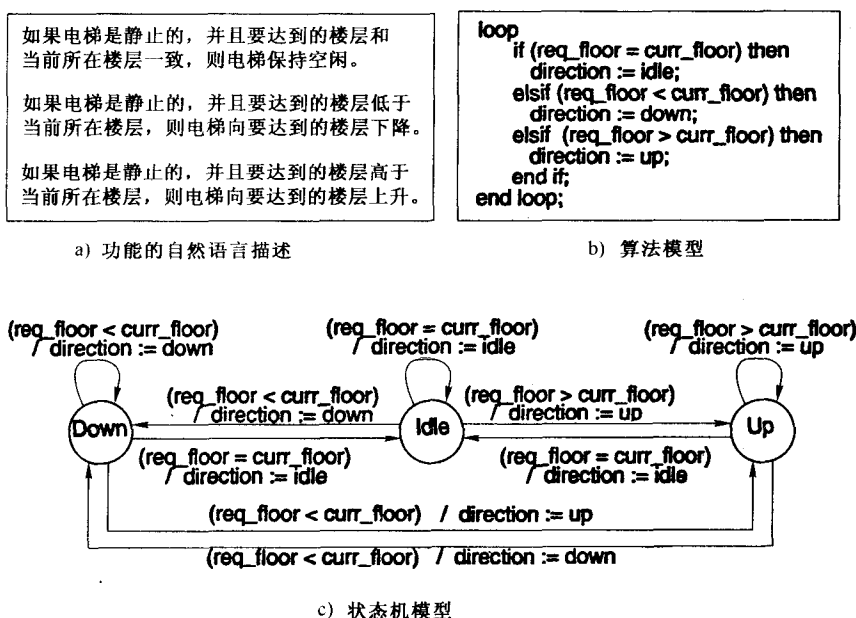


图 2-1 电梯控制器的概念视图

们感兴趣的方面。例如,在系统描述阶段,设计者除了系统的功能性之外对别的一无所知,所以他很倾向于使用一种不反应任何实现信息的模型。但是在系统的实现阶段,当有关系统组件的信息都是已知的情况下,设计者将会选取能够表示系统结构的模型。

不同的应用范围同样需要不同的模型。比如说,设计者会对实时系统和数据库系统使用不同的模型,因为前者强调的是时态行为,而后者着重于数据组织。

一旦设计者找到了可以详细指定系统功能性的合适模型,他就可以详细准确地描述出系统如何工作。但是到了这一步,设计过程还没有完成,因为这样的模型并没有准确地描述出系统将会如何制造。因此,下一步就是要将模型转化成体系结构(architecture),通过指定组件的数量和类型以及组件之间的关联,来定义模型的实现方法。例如,图 2-2 画出两种不同的体系结构,其中的任何一种都可以用来实现图 2-1c 中电梯控制器的状态机模型。图 2-2a 中的体系结构是寄存器级的实现,它用一个状态寄存器来存放当前状态,用组合逻辑来实现状态转移和输出信号的赋值。在图 2-2b 中是系统级实现,它将该状态机模型映射成软件,使用程序中的一个变量来表示当前状态,使用程序中的语句来计算状态转移和输出信号的值。在这个体系结构中,程序存放在内存中,由处理器执行。

模型和体系结构是抽象的最高层次的概念视图和实现视图。用模型来描述一个系统是如何工作的,而

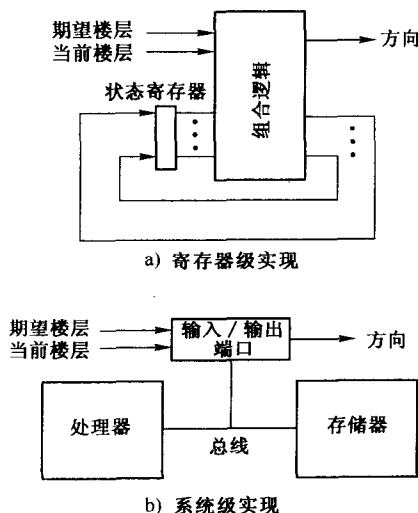


图 2-2 所使用的体系结构

体系结构用来表示系统是如何实现的。设计过程就是将模型转化成体系结构的设计任务的集合。在设计过程的最初阶段,知道的仅仅是系统的功能性。而后,设计者的工作就是要用某种基于最合适的模型的某种语言将功能性描述出来。随着设计过程的继续,过程中的每一步都加入了更多的细节,从而得到体系结构。通常,设计者们将会发现某些体系结构在实现某些模型时会更加有效。另外,设计和实现技术也会对体系结构的选取产生很大的影响。因此,在设计过程完成之前,设计者不得不考虑多种不同的实现手段与方法。

本章第一部分比较当前用于硬件和软件的设计方法学中的不同的模型,并展示每个模型中如何表示系统中的数据、状态、动作和/或结构。本章第二部分纵观在这些模型的实现中可能会用到那些不同的体系结构。

2.2 模型分类

系统设计者们在他们各种不同的硬件或软件的设计方法学中使用多种不同的模型。总体来说,这些模型可以分成5种不同的类别:(1)面向状态、(2)面向活动、(3)面向结构、(4)面向数据和(5)异构。面向状态的模型(state-oriented model),比如说有限状态机,是用一组状态以及由外部事件触发的状态间的迁移来表示系统的。面向状态的模型最适用于控制系统,比如说实时反应系统,因为在这样的系统中系统的时态行为是设计的最重要的方面。面向活动的模型(activity-oriented model),比如说数据流图,它把系统描绘成一组与数据或者执行的相关性有关的活动的集合。这个模型最适合应用到变换系统中去,比如说数字信号处理系统,在这个系统中数据以固定的速率传输并经过一系列的变换。而使用面向结构的模型(structure-oriented model),比如说框图,可以描述系统的物理模块以及它们之间的互联关系。与面向状态和面向活动的模型主要反应系统的功能性不同的是,面向结构的模型着重强调了系统的物理构成。此外,当我们想把系统表示成为一组由属性、类成员等表征的数据集合时,还可以使用面向数据的模型(data-oriented model),比如说实体-关系图。这个模型最适用于信息系统,比如说数据库系统,因为在数据库中系统的功能不及系统的数据组织重要。最后,设计者也可以使用异构模型(heterogeneous model)——综合了前4种模型的特征的一种模型——设计者可以利用它来表达出一个复杂系统的多种不同的视图。

19

同样值得注意的是一些方法学中同时使用了多种不同的模型,以此来表达系统中不同的正交视图。例如,Statemate工具[HLN⁺88]就混合了3种不同的模型:(1)功能分解和信息流的活动图;(2)时态行为和控制关系的状态图;(3)物理结构分解和信息流的方框图。应该注意这种含有不同视图的复合设计模型并不是一个异构模型,因为这3个不同的模型所表现的信息并不能够通过一个共有的数据结构而彼此互相联系。相反地,对于异构模型,我们看到是单一的、紧密集成的模型,在这个模型中不同的设计视图均来自同一个信息模型。比如说,可以考虑控制/数据流图。因为它在一种表现形式的基础上给出了两种不同的视图,可以认为它是一个异构模型。

2.3 面向状态的模型

2.3.1 有限状态机

有限状态机(finite-state machine, FSM)是面向状态模型的一个例子。它在描述控制系

统方面是最为流行的模型,因为控制系统的时态行为被表示成状态及状态之间的迁移是最为自然的。

20

一般来说,FSM 模型是由状态(state)集合、状态之间的迁移(transition)集合和与这些状态或者迁移相关的活动(action)集合所组成的。更形式化表示,FSM 是由五部分组成:

$$\langle S, I, O, f : S \times I \rightarrow S, h : S \times I \rightarrow O \rangle \quad (2-1)$$

其中, $S = \{s_1, s_2, \dots, s_l\}$ 是状态集合, $I = \{i_1, i_2, \dots, i_m\}$ 是输入集合, $O = \{o_1, o_2, \dots, o_n\}$ 是输出集合; f 是下一状态函数,根据当前状态和输入来决定下一状态;而 h 是输出函数,同样根据当前状态和输入来得到输出。值得注意的是,任何一个 FSM 模型都有一个状态作为起始状态,有一组状态作为终止状态。

在图 2-3 中,我们可以看到一个三层建筑的电梯控制器的 FSM 模型。在这个模型里,输入的集合 $I = \{r1, r2, r3\}$ 表示请停楼层。比如说, $r2$ 就表示第二层请停。输出的集合 $O = \{d2, d1, n, u1, u2\}$ 表示电梯运行的方向和所走的楼层数。比如说, $d2$ 意味着电梯应该向下 2 层, $u2$ 意味着电梯应该向上 2 层,而 n 意味着电梯应在原位置保持待命。在图 2-3 中可以看到,如果当前楼层是第 2 层(也就是说,当前状态是 S_2),并且楼层 1 请停,那么输出就为 $d1$ 。

21

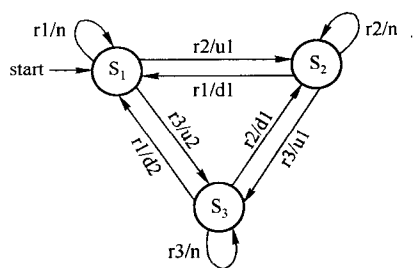


图 2-3 电梯控制器的 FSM 模型

有两种类型的 FSM 是人们极为熟悉的:即基于迁移的 FSM(Mealy 型)和基于状态的 FSM(Moore 型),它们最主要的区别就在于对输出函数 h 的定义上。在基于迁移的 FSM 中,输出值依赖于状态和输入值($h: S \times I \rightarrow O$);但是在基于状态的 FSM 中,输出值仅仅依赖于 FSM 的状态($h: S \rightarrow O$)。换句话说,在基于迁移的 FSM 中输出与迁移有关,而在基于状态的 FSM 中输出与状态有关。应该注意到图 2-3 中使用了基于迁移的 FSM 来对电梯控制器建模。相反的,图 2-4 中给出了同一个电梯控制器的基于状态的 FSM 模型,在该模型中每个状态都表明了输出值。

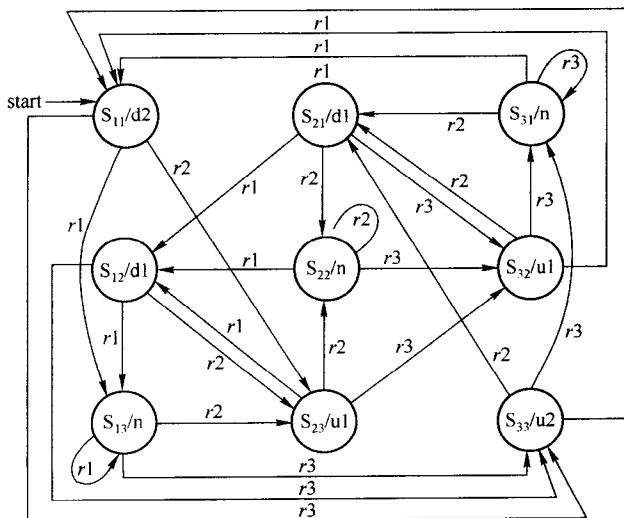


图 2-4 电梯控制器的基于状态的 FSM 模型

在实际情况下,这两种模型的主要区别在于基于状态的 FSM 可能会比基于迁移的 FSM 需要更多一些的状态。这是因为在基于迁移的模型中,多条弧可以指向同一个状态,其中每个弧有着不同的输出值;而在基于状态的模型中,每个不同的输出值都需要它自己的状态,正如图 2-4 中所示。

在 FSM 必须表示整数或者浮点数的情况下,会遇到状态爆炸的问题。因为如果一个数的每个可能的值都需要它自己的状态的话,那么 FSM 将会需要数目巨大的状态。比如说,一个 16 位的整数可以表示 2^{16} 或者 65536 个不同的状态。然而有一个相对简单的方法来消除状态爆炸问题,因为用整数或者浮点变量来扩展 FSM 是可能的,因此每个变量可以表示成千的状态。比如说,引入一个 16 位的变量就可以将 FSM 中的状态数减少 65536 个。

这种扩展的 FSM 被称为带数据通路的 FSM (FSM with a datapath, FSMD),文献 [GDWL91] 将其描述如下:定义一组存储变量 VAR , 一组表达式 $EXP = \{f(x, y, z, \dots) \mid x, y, z, \dots \in VAR\}$, 以及一组存储赋值语句 $A = \{X \leftarrow e \mid X \in VAR, e \in EXP\}$ 。而后,定义一组状态表达式 $STAT = \{Rel(a, b) \mid a, b \in EXP\}$ 来表示集合 EXP 中两个表达式之间的逻辑关系。有了这些定义,FSMD 就可以被定义为如下的五元组。

$$\langle S, IUSTAT, OUA, f, h \rangle \quad (2-2)$$

其中,输入值的集合经扩展可以包含状态表达式,输出集合经扩展已包含存储赋值语句,而 f 和 h 分别被定义为 $S \times (IUSTAT) \rightarrow S$ 和 $S \times (IUSTAT) \rightarrow (OUA)$ 之间的映射。使用这种 FSMD,我们可以将图 2-3 中的电梯控制器例子模型化成只有一个状态,如图 2-5 中所示。这样的状态数目的减少是可行的,因为我们定义了一个变量 $curr_floor$ 来存储当前楼层的值,这样就消除了为每个楼层都分配一个状态的必要。

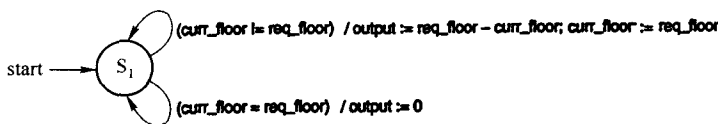


图 2-5 电梯控制器的 FSMD 模型

一般来说,FSM 适合于对那些控制占主导地位的系统建模,而 FSMD 既适合于对控制占主导地位的系统建模,又适合于对计算占主导地位的系统建模。然而,必须指出的是,FSM 和 FSMD 都不适合于复杂系统,因为这两者中没有一个可以显式支持并发性和层次性。若没有对并发性的显式支持,复杂系统将会酿成状态数目的爆炸。比如说可以考虑一个由两个并发子系统所组成的系统,每个子系统含有 100 个可能的状态。如果我们试图把此系统表示成单一的 FSM 或者 FSMD 的话,我们就必须表示系统所有的可能的状态,那就有 $100 \times 100 = 10\,000$ 个。同时,由于缺少层次性,则将会导致弧的数目的增加。举个例子,如果有 100 个状态,每个状态都要求有自己的弧来表示在特定的输入值下到指定的状态迁移,这样我们就需要 100 条弧,相反地,若模型可以层次化地将这 100 个状态分组到一个状态中,就只需要一条弧。当然,这类模型的问题是当它们多到几百个状态或者弧的情况下,人们就很难理解它们了。

2.3.2 Petri 网

Petri 网(Petri net)[Pet81, Rei92]模型是另一种类型的面向状态的模型,它特别适用于对那些由交互的并发任务所构成的系统建模。Petri 网模型由位置(place)集、迁移(transition)集

以及标记(token)集构成。标记放置于位置中,一旦迁移发生,它就在 Petri 网中流动并被接收或是产生。

更形式化表示,Petri 网是一个五元组:

$$\langle P, T, I, O, u \rangle \quad (2-3)$$

其中, $P = \{p_1, p_2, \dots, p_m\}$ 是一组位置, $T = \{t_1, t_2, \dots, t_n\}$ 是一组迁移, 并且 P 和 T 没有交集。而后, 输入函数 $I: T \rightarrow P^+$, 定义了为迁移提供输入的所有的位置, 而输出函数 $O: T \rightarrow P^+$, 定义了每一个迁移的所有输出位置。换句话说, 输入和输出函数明确定义了位置和迁移之间的关联关系。最后, 标记函数 (marking function) $u: P \rightarrow N$ 定义了每个位置中标记的个数, 其中 N 是非负整数集合。

在图 2-6 中, 我们看到 Petri 网的图形表示和文字表示。注意在这个 Petri 网中有五个位置 (图中表示为圆形) 和四个迁移 (图中表示为竖线段)。在这个例子中, 位置 p_2, p_3 和 p_5 给迁移 t_2 提供了输入, 而 p_3, p_5 是 t_2 的输出位置。标记函数 u 给 p_1, p_2, p_5 各分配了一个标记, 给 p_3 分配了两个标记, 因此还可以表示为 $u(p_1, p_2, p_3, p_4, p_5) = (1, 1, 2, 0, 1)$ 。

如上面所提到的, Petri 网是通过触发 (firing) 各个迁移来执行操作的。一个

迁移只有在准备好的情况下才可以被触发——也就是说, 它的每一个输入位置中至少含有一个标记。当一个迁移将它的输入位置中所有的标记都转移走, 并且为每一个它的输出位置都放入一个标记后, 我们就说迁移被触发了。比如说在图 2-6 中, 在迁移 t_2 被触发后, 标记函数 u 就变为 $(1, 0, 2, 0, 1)$ 。

Petri 网之所以有用是因为它们可以有效地模型化系统的多种特征。比如说, 图 2-7a 显示顺序性 (sequencing) 模型, 在该模型中迁移 t_1 在迁移 t_2 后触发。在图 2-7b 中是非确定性 (non-deterministic) 分支的模型, 其中两个迁移都是准备好的但是只有一个才会触发。在图 2-7c 中是一个同步 (synchronization) 模型, 一个迁移只有当它的两个输入位置都含有标记时才会触发。图 2-7d 给出了建立资源竞争 (resource contention) 模型的方法, 其中两个迁移竞争放置在中间的位置之中的同一个标记。在图 2-7e 中, 可以看到如何建立并发性 (concurrency) 的模型, 其中两个迁移 t_2 和 t_3 可以同时触发。更确切地说, 图 2-7e 建立了两个并发的进程模型: 产生者和消费者; 位于中间的位置中的符号由 t_2 产生, 由 t_3 消费。

Petri 网模型可以用来检查和验证某些实用的系统性质, 比如说安全性和活跃性。例如, 安全性 (safety), 指的是 Petri 网保证了网中的符号数量不会无限的增长的特性。事实上, 我们不能构建一个标记个数无限多的 Petri 网。另一方面, 活跃性 (liveness) 指的是 Petri 网保证至少有一个迁移可以触发, 确保其无死锁操作的特性。

虽然 Petri 网在建模和分析并发系统方面有着许多的优势, 但是它同样存在着类似于 FSM 的那些局限性: 系统复杂度的增长会快速导致系统的不可理解性。

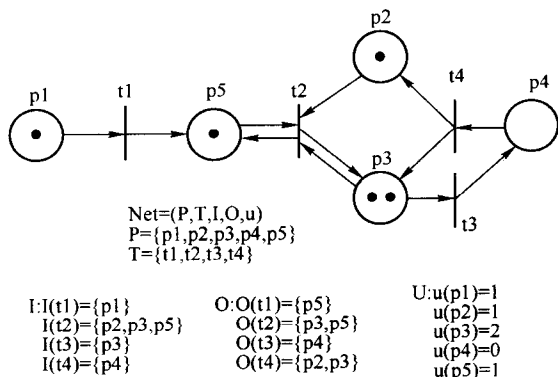


图 2-6 一个 Petri 网例子

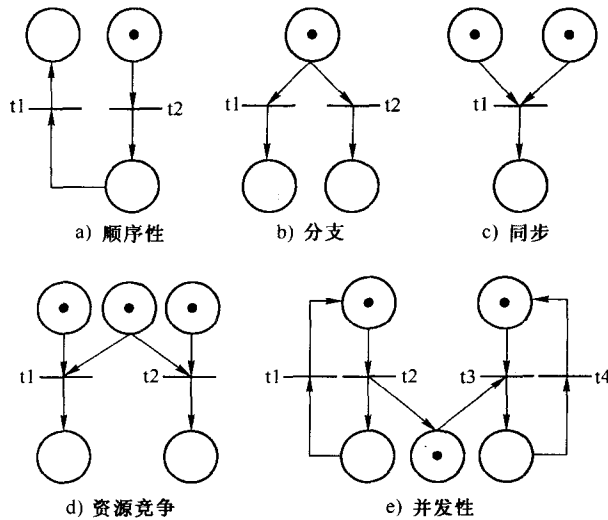


图 2-7 Petri 网的多重表示

2.3.3 层次化并发有限状态机

层次化并发有限状态机 (hierarchical concurrent finite-state machine, HCFSM) 本质上是 FSM 模型的扩展, 它在 FSM 模型中加入了对层次 (hierarchy) 和并发 (concurrency) 的支持, 因此消除了用 FSM 模型来描述层次化的和并发的系统时所发生的状态和弧的爆炸的可能性。

与 FSM 一样, HCFSM 模型由一组状态和一组迁移组成。但是与 FSM 不同的是, HCFSM 中每个状态可以进一步的分解为一组子状态, 就可以模型化层次性。而且, 每个状态也可以被分解为并发子状态, 这些子状态并行执行并通过全局变量进行通信。本模型中迁移既可以是结构化的也可以是非结构化的, 结构化的迁移只能在同一层次上的两个状态间进行, 而非结构化的迁移可以发生在任意的两个状态之间而不管它们的层次关系。

一种极其适合于 HCFSM 模型的语言是 Statecharts [Har87], 因为它可以轻松支持并发状态之间层次、并发和通信的概念。Statecharts 使用了非结构化的迁移和广播通信机制, 其中任一给定状态所产生的事件都会被其他所有状态所发现。

Statecharts 语言是一种图形语言。其特别的一点是用圆角矩形来表示任意层次上的状态, 用封装来表示这些状态之间的层次关系。状态间的虚线表示并发性, 箭头表示状态间的迁移, 每个箭头上都标注有事件, 并且在需要时标注有加括号的条件和/或活动。

图 2-8 给出了一个用 Statecharts 表示的系统的例子。在这张图中, 我们可以看到状态 Y 分解成了两个并发的状态, A 和 D; 前者由两个进一步分解的子状态 B 和 C 组成, 而后者由子状态 E、F 和 G 组成。图中的黑点表示状态的起始点。根据 Statecharts 语言描述, 当状态 C 中事件 b 发生,

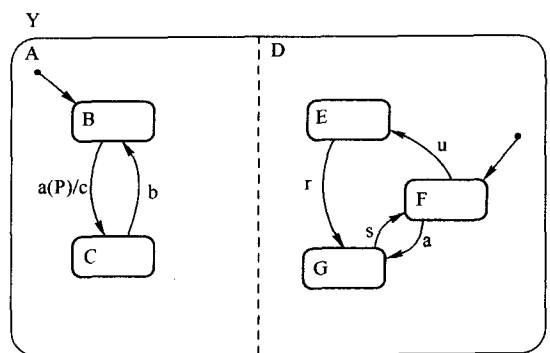


图 2-8 Statechart: 层次式的并发状态

则 A 将转移到状态 B 。另一方面,如果状态 B 中事件 a 发生,则 A 将转移到状态 C ,但是仅当条件 P 在事件发生的瞬间成立。在 B 到 C 的迁移过程中,与迁移有关的活动 c 将会被执行。

由于其层次性和并发性构造,HCFSM 模型极其适合于表示复杂控制系统。然而,像其他的面向状态的模型一样,这个模型的问题在于它过度地集中于控制的模型化,这就意味着它只能与非常简单的活动(比如说赋值语句的迁移或状态)相关联。因此,HCFSM 并不适合于对复杂系统的某些特性建模,它们可能需要复杂的数据结构或是可能在每个状态中都会执行任意的复杂活动。对于这些系统,仅靠这个模型恐怕不能胜任。

2.4 面向活动的模型

2.4.1 数据流图

前面讨论了面向状态模型最常被用于反应系统,在这些系统中系统的状态随着外部事件而发生变化。相对地,数据流图(dataflow graphs,DFG)[DeM79, Dav83, GDWL91]则主要用于传输系统,在这些系统中输出是通过对系统输入的一组运算而得到的。因而,DFG 中没有状态和外部事件来触发状态的变化。它们只是简单地由表示数据流的弧集所连接的活动集(传输)所组成。

更确切地说,DFG 由结点(node)集和边(edge)集所组成。在数据流图中有几种不同类型的结点。第1种类型的结点包括输入结点(input node),也叫做源结点(source node)和输出结点(output node),也叫做目标结点(destination node),它们表示输入或者输出的数据。第2种类型是活动结点(activity node),也叫做进程结点(process node),它表示转移或者操控数据的活动。这样的活动可以被描述为一段程序、一个过程、一个函数或者一个指令,甚至是一个算术运算。第3种结点类型是数据存储类型(data store type),它表示数据存储的不同形式,比如说是数据库中的记录,操作系统中的文件,或者是内存或寄存器中的一个变量。这些 DFG 中的多种结点通过有向边相互连接,通常这些有向边上都标有这两个结点间正在传送的数据。因此每个活动结点都可以表示成另外的一个 DFG,因此这个模型支持层次化。

通常,在 DFG 的图形表示中,矩形表示输入或输出结点,圆形表示活动结点,不闭合的矩形表示数据存储结点。数据流表示为上面标有相关数据的弧。在图 2-9a 中给出了一个例子。这个例子中的系统由两个活动组成: A_1 和 A_2 ,后者被进一步的分解为活动 $A_{2.1}$ 、 $A_{2.2}$ 和 $A_{2.3}$ 。在这个系统中,数据 X 将会从输入流到 A_1 ,而数据 V 将会在 A_1 处计算并存储到文件中。而后数据 V' 将会从文件中取出,并与 A_1 所产生的数据 Y 一道被当作输入送

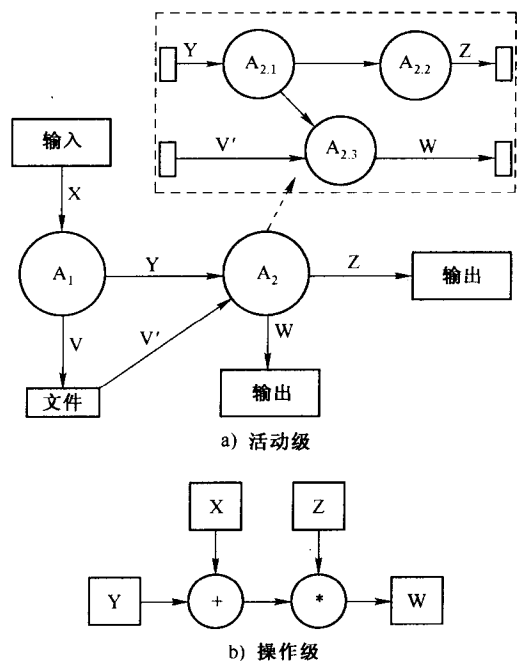


图 2-9 数据流图

给 A_2 。数据 Z 和 W 是 A_2 产生的输出。

通过将不同的对象与图中的结点和边相关联,数据流模型就可以用于不同的应用领域,或者同一领域不同的设计阶段,因此它是很有价值的。比如说,在数字信号处理领域,数据流图中的结点可以表示变量或者算术运算,例如加法和乘法,而数据流边可以用来表示数据相关性 (data dependency),如图 2-9b 中所示。在这个例子中,操作 ‘+’ 与数据 X 和 Y 相关,而操作 ‘*’ 与数据 Z 和操作 ‘+’ 的输出相关。

29

值得注意的是,在不同活动之间,除了存在问题本身的数据相关性之外,DFG 不能描述其他的强加操作顺序,而且不含有关于实施过程的任何的信息。基于这些原因,DFG 常用于系统描述阶段,作为设计者和用户之间沟通的手段。同样也应该注意到,因为 DFG 支持层次分解,所以这个模型也适用于详细说明复杂的传输系统。然而,因为这个模型除了它的数据相关性之外没有表达系统的任何的时态行为或控制活动,因此它在对内嵌系统建模方面还存在着缺陷。

30

2.4.2 流程图

流程图(flowchart)[Dav83, Sod90],也称为控制流图(control-flow graph,CFG),是一种与 DFG 在许多方面都很相似的面向活动的模型。它们的区别在于弧的意义不同。在 DFG 中弧用来表示数据流,而在流程图中则表示顺序性或者控制流。流程图与 FSMD 也很相像,这两者都强调了系统的控制方面,而区别在于对迁移的触发机制不同:FSMD 中的迁移是通过外部事件的发生来触发的,而流程图中的迁移是当一个具体活动完成时触发的。

一般来说,流程图由结点集合和弧的集合组成。有 3 类结点:第 1 类是起始结点(start node)和结束结点(end node),它们表明了流程图的开始点和结束点;第 2 类是计算结点(computation node),通过赋值语句序列定义数据的传输;第 3 类是决策结点(decision node),用于控制分支。流程图中的不同类型的结点是通过有向边来相互连接的,它们表明了结点执行的顺序。用图形表示的话,流程图使用圆角矩形来表示起始结点和结束结点,用矩形表示计算结点,而用菱形方框表示决策结点。图 2-10 给出了流程图的一个例子,它是用来计算数组 MEM 中 N 个数中的最大值。

31

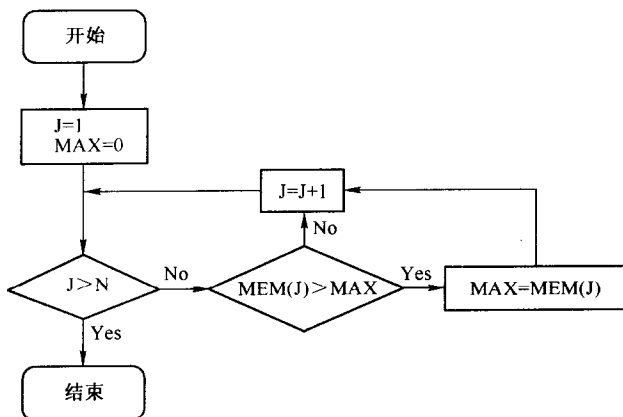


图 2-10 寻找最大值的流程图表示

当需要把系统看成一组在控制流管理下的活动序列时,流程图就很实用。这个模型适合于那些含有已定义好的并且不依赖于外部事件的任务的系统。当我们需要表示自然的数据相

关性时,该模型还可以用于强加一个特定的执行顺序给 DFG 中的活动。因为设计者强加的顺序需要系统遵循某一特定的实施方案,所以只有当系统的实施方案已确定的情况下才能使用流程图模型。

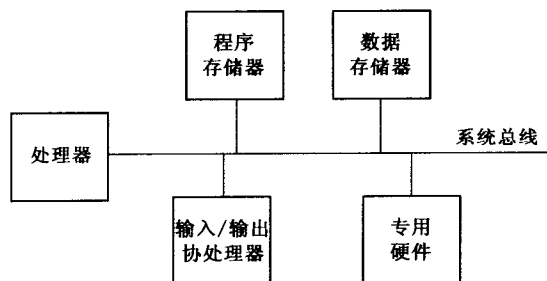
2.5 面向结构的模型

组件连接图

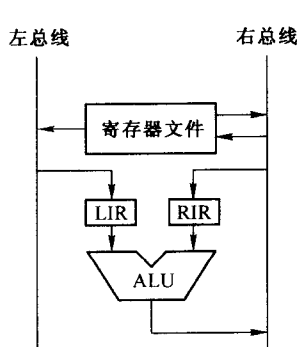
组件连接图(component-connectivity diagram, CCD)是一类面向结构的模型,它们被用于描述系统的物理结构而不是功能性。DFG 或流程图表示由数据或者控制相关性所连接的系统活动,而与它们不同的是, CCD 表示系统组件以及它们的互联关系。换句话说,它对系统的结构视图建模。

CCD 也是由结点集和边集构成。结点表示不同的组件(component),这些组件定义为结构对象,并已确定其输入集和输出集。例如门、ALU、处理器甚至是子系统都可以作为组件。而边则表示这些组件间不同的连接,比如说总线和连线。

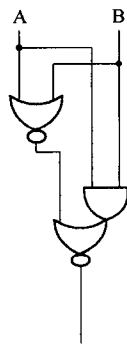
因为 CCD 允许用结点和边表示不同的对象,所以这个模型可以用于表示各种模型。比如在图 2-11 中,CCD 在三个不同的抽象层次上例化,产生系统框图(system block diagram)、寄存器级原理图(register-level schematic)和门级原理图(gate-level schematic)。在系统框图中,组件可以是系统级模块,例如处理器、存储器或 ASIC。应该注意这些组件之间的连接只有部分被



a) 系统方框图



b) 寄存器级原理图



c) 门级原理图

图 2-11 面向结构建模

表示出来,图中没有包括详细的连接信息,像数据总线的宽度和特定的控制信号等。在寄存器级结构图中,组件指的是寄存器级单元,比如 ALU、寄存器、选择器或总线,而连接则定义数据如何在这些算术和内存元素中传输。在这种类型的原理图中,通常控制信号并不表示出来。而门级原理图则使用门作为组件,即门元件。而且这些元件之间的连接表示实际的物理连线。换句话说,数据和控制连接是完全指定的。

由于组件连接模型特别适合于表示系统的结构,因此常常用于设计过程的最后一阶段,设计者需要详细描述系统的实施方案。

2.6 面向数据的模型

2.6.1 实体-关系图

到目前为止,我们讨论了面向状态的模型和面向活动的模型。面向数据的模型与它们有

着很大的区别,着重表示数据而不是表示处理数据的那些活动。面向数据的模型通常用于信息系统的设计方面,因为在这种系统中数据的组织比设计的其他方面都重要。面向数据的模型的一种表示方法是所谓实体-关系图(entity-relationship diagram, ERD)[Che77, Teo90],它将系统定义为实体以及实体之间的不同关系。

在 ERD 中,实体表示用矩形方框表示,而其关系用菱形方框表示。比如,我们现在想要表示一个百货公司为它的客户订购了哪些产品这个信息。其 ERD 模型需要四个实体:客户(Customer)、购买订单(Order)、产品(Product)及供应商(Supplier),如图 2-12 所示。进而,需要表示这些实体之间的关系:客户需要哪些产品,在什么时间给某产品供应商一份订单。这些关系遵循下列规则:可用性(Availability)指定供应商和它们所生产的产品;需求(Request)将客户和他们所需要的产品联系起来;产品订单将一种特定的产品和它的特定的供应商和客户订单联系起来。

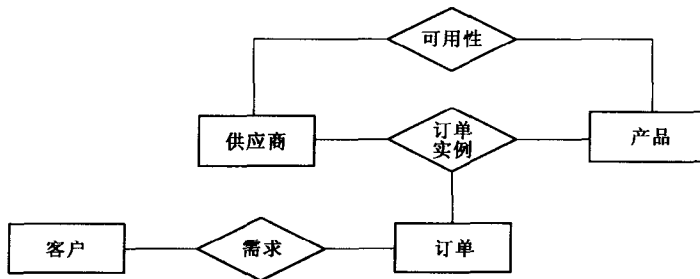


图 2-12 一个实体-关系图的例子

在这种图中,每个实体都有一个唯一类型的数据,并拥有一种或多种指定的属性。比如在百货公司销售这个例子里,实体“客户”的属性可能是每个客户的姓名和地址,而与实体“产品”相关联的属性则是每个产品的品名和价格。作为规则,每个关系都显示出了它所关联的实体的某些“事实”,比如说关系“订单实例”就给出了这样的信息:哪个客户想要哪个供应商的什么产品。

因为 ERD 给出了整个系统的一个很好的数据视图,所以这个模型特别适合于想要组织多种数据之间的复杂关系的时候。但是我们也应该注意到 ERD 模型不能够描述系统的任何功能或时序行为。

34

2.6.2 Jackson 图

另外一种面向数据的模型是 Jackson 图(Jackson's diagram)[Sut88],它有着自己独特的优点。实体-关系图只倾向于强调数据的属性和其相互关系,而 Jackson 图对每一个数据建模的方法是:利用数据的结构,把数据分解成子数据。比如说, Jackson 图很适合于对一个含有多个子属性的记录建模。事实上, Jackson 图适合于那些需要进行分解的数据,而这些数据的分解方式会比那些适合简单记录的方式要更复杂。

Jackson 图中的数据通过树形结构进行分解,在树形结构中叶子结点的基本数据类型,非叶子结点是通过多种操作像组合(composition)(AND)、选择(selection)(OR)和迭代(iteration)(*)而得到的复合数据类型。组合操作产生了包含有两种或者多种子类型的一种新的数据类型。而选择操作通过在这些子类型中选择一种来生成它自己的数据。迭代操作则通过复制它

35 的子类型中的某些元素来生成数据。

图 2-13 给出一个 Jackson 图的例子,为画面对象(drawing object)建立模型。在这张图中,画面表示一个复合数据类型,由颜色、形状和由“*”操作表示的多个用户所构成。此外,每个用户类型由姓名属性组成,而每个形状类型为圆或者矩形。最后,矩形类型本身由两个子类型组成:宽度和高度,而圆类型只有一个子类型:半径。

与 ERD 模型不同的是:ERD 最适合于表示有复杂关系的数据,Jackson 图则最适合于表示有着复杂复合结构的数据。但是也应该注意到,这个模型的局限性和 ERD 很类似,也不能描述系统的功能性或时序行为。

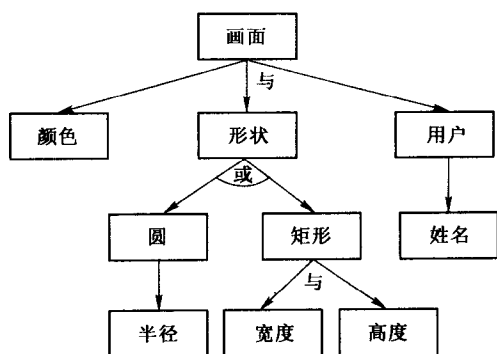


图 2-13 Jackson 图

2.7 异构模型

2.7.1 控制/数据流图

36 控制/数据流图(control/data flow graph,CDFG)[OG86, LG88]是一种异构模型,结合了流程图 CFG 和数据流图 DFG 两种模型的优点。换句话说,CDFG 既含有用于表示活动之间的数据流的 DFG,又含有表示 DFG 的顺序关系的 CFG。因此,CDFG 模型可以在一个表示中
37 既可以显式表示数据相关性,又可以显式表示系统的控制顺序。

图 2-14b 是图 2-14a 中描述的程序的 CDFG 表示。应该注意程序语言中的控制结构,比如 case 语句,被映射成了控制流结点,控制流构造之间的赋值语句组被表示成了 DFG。同样值得注意的是,CFG 和 DFG 用虚线连接起来,这表示 DFG 中的哪个活动和 CFG 中的哪个结点有关。一旦控制转移到了某个结点,就会执行与其相关联的活动。

然而,CDFG 并不局限于表示程序设计语言中的控制构造和赋值语句。相反地,它也被用于表示系统需求的任何复杂活动和控制行为,并且也常用于实时系统的设计。例如,文献[WM85]中提出的模型是由 DFG 加上 CFG 组成的,其中 CFG 用状态机模型表示。在这个系统中,CFG 可以响应外部和内部事件,也可以通过产生控制行为来控制 DFG 的执行,比如说可以开始一个活动或者停止一个活动。

38 在图 2-15 中,我们看到了一个活动级别 CDFG 的例子。根据这个描述,当 CFG 处于状态 S_0 时并且开始有事件发生,则活动 A_1 和 A_2 就被执行,系统进入状态 S_1 。另一方面,当 CFG 处于状态 S_1 并且有事件 $W=10$ 发生,则活动 A_1 被停止,活动 A_3 被执行,系统状态迁移到 S_2 。最后,当系统处于状态 S_2 而且事件停止发生,那么活动 A_2 和 A_3 被停止,系统又回到状态 S_0 。 W 、 X 、 Y 和 Z 表示 DFG 中表示的多个活动之间的数据流动。

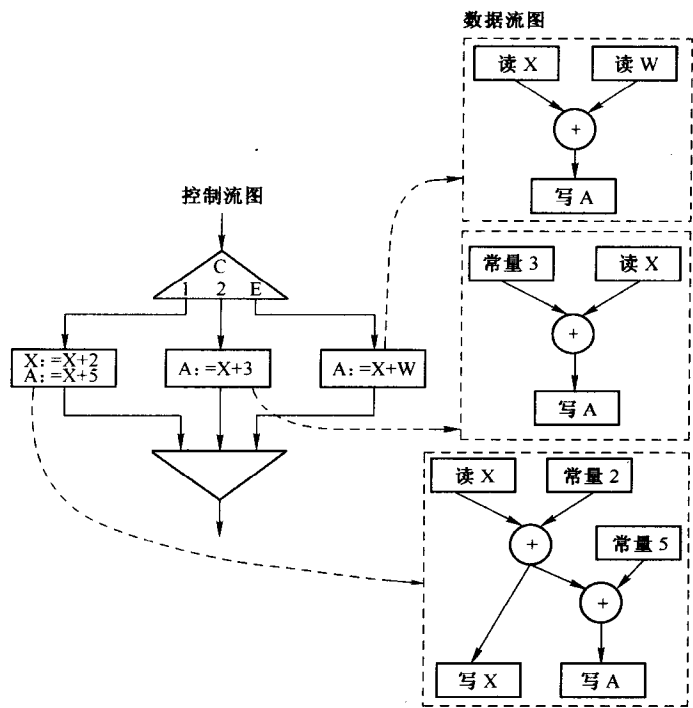
CDFG 的主要的优点就在于它改进了 DFG 不能够表示系统控制的不足,也改进了 CFG 不能表示数据相关性这个不足。总之,它近乎完美且能够很好的适用于许多不同的设计领域,比如说实时系统和 ASIC[GDWL91]的行为综合。

2.7.2 结构图

由 Yourdon 和 Constantine 提出的结构图(structure chart)模型[YC78]是另一种异构模

```
case Cis
  when 1 =>    X:=X+2;
               A:=X+5;
  when 2 =>    A:=X+3;
  when others => A:=X+W;
end case;
```

a) 程序代码



b) CDFG

图 2-14 控制/数据流图

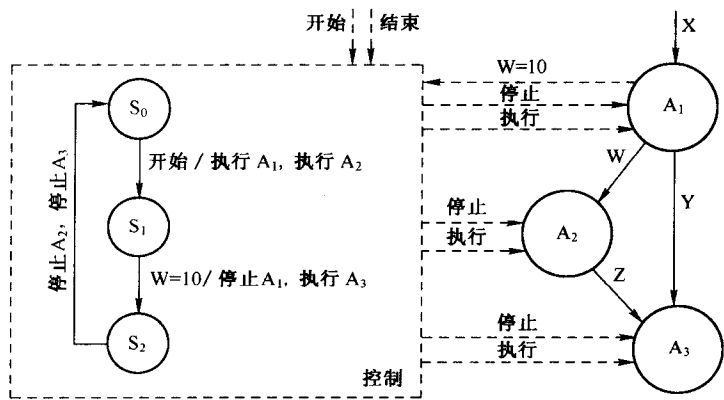


图 2-15 活动级 CDFG

型,用于在同一个表示中说明系统的数据、活动和控制访问。因为其易于理解的特点,这种结构图对于程序设计者来说是很有用的。

结构图是由结点集和边集组成。结点表示活动,边表示程序设计语言中的过程调用或者函数调用。边上标出了活动中传递的数据。而活动本身的执行控制是通过控制结构集来描述的,包括分支(branch)、迭代(iteration)和子程序调用(subroutine call)。

图形表示的话,系统中的所有的活动都用矩形表示,而这些活动中交换的数据用带标号的箭头表示。结构图进一步用弧表示活动之间的过程调用,用菱形符号表示分支结构,用自闭合弧表示迭代结构。

图 2-16 是结构图的一个例子。对于活动的顺序,模块 Main 首先调用模块 Get 来获得数据 A 和 B。Get 再调用 Get_A 和 Get_B。然后,模块 Main 将数据 A 和数据 B 传送到变换模块,而变换模块则会根据某些条件,或者调用 Change_A 将 A 传送到 A',或者调用 Change_B 模块将 B 传送到 B'。当从传送模块 Transform 获得 A'和 B'后,模块 Main 就会将这些数据送到 Compute,它将会调用两个循环模块:Loop1 和 Loop2。当这些模块完成了各自的任務后,Compute 模块就将数据 C 和 D 返回给模块 Main,进而 Main 将数据 C 送给 Out_C。

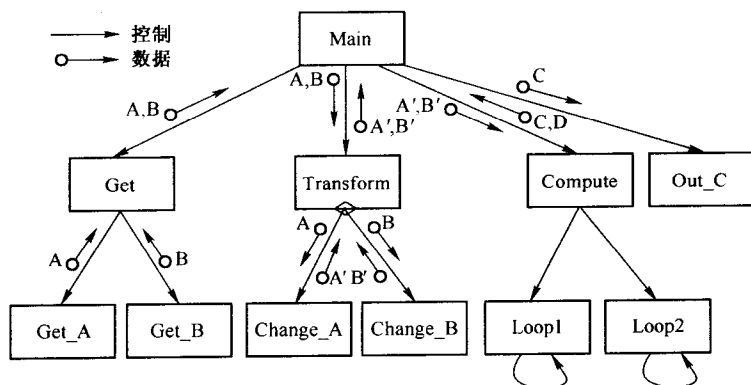


图 2-16 一个结构图的例子

如同 CDFG 中的情况,结构图既可以表示控制信息又可以表示数据信息。但是,应该注意到结构图并没有完全说明执行顺序。比如说,我们不知道 Get_A 和 Get_B 被调用时的顺序,这些模块的执行顺序是由数据相关性来约束的。另一方面,结构图可以表示模块之间分支、迭代和过程调用上强加的控制。由于它可以在某种程度上描述执行的顺序,因此结构图主要用于设计顺序程序的预备阶段。

2.7.3 程序设计语言模式

与结构图一样,程序设计语言(programming language)也可以作为支持数据、活动和控制建模的异构模型。但是与结构图不同的是,程序设计语言使用的是文本表示形式而不是图形形式。

程序设计语言有两种主要的类型:命令式语言(imperative language)和声明式语言(declarative language)。命令式语言包括像 C 和 Pascal 这样的语言,它们使用执行的控制驱动模型,其中的语句按照程序中的顺序进行执行。而 LISP 和 PROLOG 属于声明式语言,因为它们通

过需求驱动或者模式驱动的计算来对执行建模。这两种类型的语言的主要区别在于:声明式语言没有显式给出执行顺序,而是着重于通过函数集或者逻辑规则来定义计算的目标。

在数据建模方面,类似于 Jackson 图,命令式程序设计语言也提供了多种数据结构,包括:基本数据类型,如整型和实型;复合数据类型,如数组和记录。程序设计语言通过语句来对小型活动建模,而将大型活动通过函数或过程建模。函数或过程同时作为支持系统层次化的一种机制。这些程序设计语言通过控制结构详细规定了活动执行顺序,建立控制流模型。控制结构包括顺序组合(常用分号标识)、分支(if 和 case 语句)、循环(while、for 和 repeat)以及子程序调用。

使用命令式程序设计语言的优点在于它很适合于对计算占主导地位的行为建模。在这些行为中一些问题是通过算法的运用来解决的,比如说我们想要对一个数组中的各个数进行排序。

一些程序设计语言,比如像通信顺序进程(communicating sequential process, CSP)[Hoa78, Hoa85]、ADA 和 VHDL,是为了支持并发执行而发展起来的。这些语言使用不同的通信机制。在 CSP 中,并发进程之间的通信采用消息传递(message passing)方式,数据从一个进程经由通信通道传至另一个进程。在 ADA 中,并发任务通信是利用约会(rendezvous)方式实现,每个任务需要在某个同步点上等待其他任务传来的数据。最后,在 VHDL 中并发进程之间通信通过共享内存(shared memory)实现,即不同的进程对全局信号进行读写操作。

41

虽然程序设计语言非常适合对系统的数据、活动和控制机制建模,但是其最大问题在于并没有显式对系统状态建模,对于嵌入式系统的建模是个不利因素。

2.7.4 面向对象的模型

面向对象的模型(object-oriented model)[Boo91]是由面向数据的模型发展而来,它的特点是趋向于将系统看成一组对象(object)。对于这个模型,每个对象是由一组数据(data)和一组变换数据的操作(operation)组成的。在某些方面,面向对象的模型很像由实体组成的实时系统,因为每个系统中的实体通常含有一组定义好的操作,实体可以通过这些操作与其他的实体进行交互作用,来完成其他实体要求的服务。

面向对象模型有几个其他模型不具有的特点。首先,它们能通过将数据封装在对象中并且使数据对于其他对象不可见来支持数据抽象(data abstraction)或者信息隐藏(information hiding)。这个特点意味着当一个对象想要改变其他对象中的数据时,前者必须请求后者的服务。因为此项要求,一个对象中的内部变化不会影响到系统中的其他对象。除此之外,面向对象模型同样能够用一种自然的方式来表示并发性(concurrency),因为任何一个对象都是作为系统的一个组成部分被创建并且要在它的生存期内与其他对象共同存在,同时还要执行独立于其他对象的任务。

在图 2-17 中,我们可以看到,除了它的对象以外,系统还要包括一组转换,用于定义随时间变化的系统行为。正如上面所说,这些迁移不能直接改变数据,只能由每个对象来请求这些改变所需要的特定操作。

很明显,面向对象的模型有许多优点。然而,系统设计者有时会发现对于需要复杂转换函数的系统来说,这个模型有些不尽如人意,除非再补充一些可以足够表示这些转换的其他模型。

42

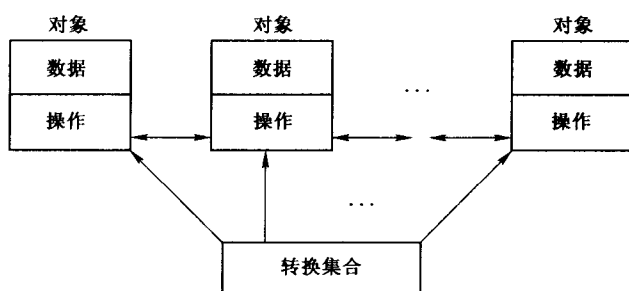


图 2-17 带有对象和转换函数的系统

2.7.5 程序状态机

程序状态机(program-state machine, PSM)[GVN94, VNG91]是另一种异构模型,它综合了 HCFSM 和程序设计语言范式。这个模型基本上由层次化的程序状态(program-state)组成,其中每个程序状态表示一个计算的独特的模型。在任何给定的时间里,只有程序状态的一个子集处于活跃状态,比如,执行它们的计算任务。

在这一模型的层次结构中,既含有复合程序状态又含有叶子程序状态。复合程序状态可以进一步被分解为或者并发或者顺序的程序子状态。如果是并发的,那么只要程序状态处于活跃状态,则其所有的程序子状态也是活跃的状态。如果是顺序的,则当程序状态活跃时,程序子状态只是活跃一次。顺序分解的程序状态包含一组迁移弧,它们表示这些程序子状态间的顺序关系。一共存在两种迁移弧。第 1 种,完成时迁移弧(transition-on-completion arc, TOC),只有当源程序子状态完成了它的工作以及相关的弧的条件成立的时候,它才进行迁移。第 2 种,立即迁移弧(transition-immediately arc, TI),无论何时只要弧的条件成立就可以进行迁移,而不必考虑源程序子状态是否完成了计算。在层次结构的最低层,是叶子程序状态,它的计算是通过程序设计语言语句进行描述的。

当采用程序状态机模型时,系统作为一个实体图形化地表示为矩形方框,而实体中的程序状态表示成带圆角的矩形方框。程序子状态之间的并发关系用虚线表示。迁移用有向箭头表示。起始状态用三角形表示,单个程序状态的完成用一条指向完成点的迁移弧来表示,完成点用一个放在状态中的小方块来表示。TOC 弧起始于源子状态中的小方框,而 TI 弧则起始于源子状态的边界。

图 2-18 给出了一个程序状态机的例子,它由根状态 Y 组成,而 Y 本身又是由两个并发子状态 A 和 D 构成。状态 A 包含两个顺序子状态 B 和 C。虽然图中只给出了状态 D 的程序,但是应该注意到状态 B、C 和 D 都是叶子状态。根据上面给出的图形符号,我们可以看到标有 e1 和 e3 的弧是 TOC 弧,而标有 e2 的弧是 TI 弧。弧的配置可能意味着当状态 B 结束而且条件 e1 成立时,控制将转移到状态 C。然而,如果条件 e2 在状态 C 中成立时,则不管 C 是否完成都会将控制转移到状态 B。

由于 PSM 可以在单一模型里表示系统的状态、数据和活动,所以它比 HCFSM 更适合于对那些有着复杂数据和活动且与每个状态相关的那些系统建模。PSM 还可以克服程序设计语言的最大障碍,因为它可以显式表示状态。它允许设计者使用层次化状态分解方法来描述系统,直到觉得使用程序结构比较方便。程序设计语言模型和 HCFSM 模型是 PSM 模型的

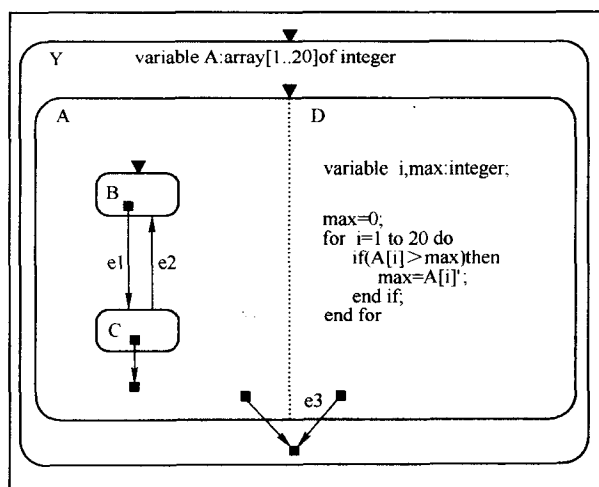


图 2-18 程序状态机例子

两个极端:程序可以看成是只有一个叶子状态包含语言结构的 PSM 模型;HCFSM 可以看成是 PSM 模型中的所有叶子状态都不含有语言结构。

2.7.6 队列模型

队列模型(queueing model)[Gif78]与我们前面提到的模型有一些不同之处,主要在于那些模型是用于系统设计的,而队列模型是用于分析系统的。比如说,当我们想要找到系统性能或者资源的瓶颈的时候,就可以使用队列模型。

队列模型的特别之处在于它将系统描述成队列(queue)和服务端(server)的一个网络。到达的请求都被存放在队列之中等待着服务器的处理。在图 2-19a 和 b 的例子中,分别为有一个服务器和多个服务器的队列模型。

队列模型的价值在于它们提供了对于解决系统问题很有必要的数学分析的基础。比如说,如果我们知道系统的某些特征,像服务器的数目、队列的类型、两个连续到达的请求之间的时间间隔以及一个请求需要的服务时间,那么队列模型可以提供更进一步的信息,像利用率(服务器繁忙时间所占比例)、队列长度(队列中等待的平均请求数目)和吞吐率(请求通过服务器的速率)等。设计者们可以利用这些新的信息来定位系统中的瓶颈位置。

应该注意到,不同的模型需要不用类型的数学分析。某些模型,比如说只有单一服务器和单一队列的模型,只需要相对简单的技术,而其余的模型则需要更为复杂的技术,更有一些模型几乎不太可能作定量的分析。

作为实例,下面分析有单一服务器和单一队列的模型。为了进行数学分析来确定系统的行为,必须了解模型的几个参数。比如,必须知道两个连续到达的请求之间的时间间隔,也要

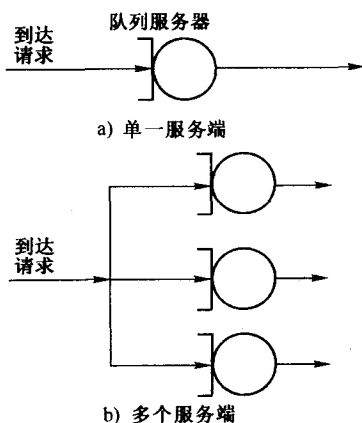


图 2-19 队列模型

- 46 知道每个请求所需要的服务时间。用适当分布的随机变量建立这两个时间模型是可能的,正如这个例子中相互到达的时间和服务时间假设成指数分布。在这个例子里,如果平均的到达时间间隔为 λ , 而且平均服务时间为 μ , 当 $\lambda > \mu$, 就可以到达一个稳定的队列, 因为它可以避免有太多的请求在队列里等候。然而, 当 $\lambda \leq \mu$, 队列将会不稳定, 从而会导致无限的等待时间。读者可以参考[Gif78, Laz84]中关于队列模型的进一步的分析技术。

2.8 体系结构分类

目前为止, 我们已经阐明了各种模型如何描述系统的功能性、数据、控制和结构。这里再介绍作为描述模型的体系结构, 它详细说明系统在实际中是如何实现的。体系结构的作用就是描述组件的数量、每个组件的类型和这些系统中不同组件之间所有连接的类型。

体系结构的范围可以从简单的控制器一直到大规模并行处理器。然而, 尽管存在多样性, 体系结构还是可以分成几个不同的类别, 即(1)专用体系结构(application-specific architecture), 如 DSP 系统; (2)通用处理器(general-purpose processor), 如 RISC; (3)并行处理器(parallel processor), 如 SIMD 机和 MIMD 机。

2.9 专用体系结构

2.9.1 控制器体系结构

最简单的专用体系结构就是控制器类, 它直接实现 2.3.1 节讲到的用五元组 $\langle S, I, O, f, h \rangle$ 定义的有限状态机模型。如图 2-20 所示, 控制器由一个寄存器和两个组合模块组成。这个寄存器, 通常叫做状态寄存器, 用来存储 S 中的状态, 而两个组合模块, 分别表示次态函数和输出函数, 实现函数 f 和 h 。输入和输出则表示由集合 I 和 O 定义的布尔信号。

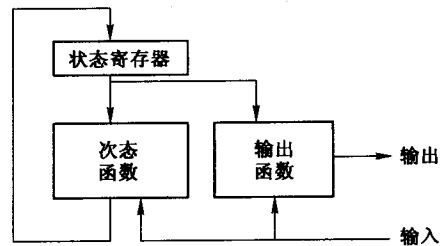


图 2-20 通用 FSM 框图

如前所述, 存在着两种不同类型的控制器, 它们是基于迁移的和基于状态的。这些控制器类型不同之处在于它们是如何定义输出函数 h 的。对于基于迁移的控制器, h 被定义为一个 $S \times I \rightarrow O$ 的映射, 这就意味着输出函数和两个参数有关, 即状态寄存器和输入。而另一方面, 对于基于状态的控制器, h 被定义为 $S \rightarrow O$ 的映射, 这就意味着输出函数只依赖于一个参数——状态寄存器。因为输入和输出都是布尔信号, 所以对于这两种情况, 这个体系结构很适合于实现那些不需要复杂数据操作的控制器。控制器综合包括状态最小化、编码、布尔最小化, 以及对次态函数和输出函数的映射技术。

2.9.2 数据通路体系结构

数据通路体系结构用于许多需要在不同的数据集上重复执行固定计算的应用中, 比如说在数字信号处理(digital signal processing, DSP)系统中用于数字滤波、图像处理和视频压缩。数据通路体系结构常常由高速的算术单元组成, 而且通过并行连接并大量采用流水线技术来达到高的吞吐率。

- 48 在图 2-21 中, 我们可以看到两个不同的数据通路, 分别用于实现有限脉冲响应(finite-im-

pulse-response)滤波器(FIR),它由下面表达式定义。

$$y(i) = \sum_{k=0}^{N-1} x(i-k)b(k)$$

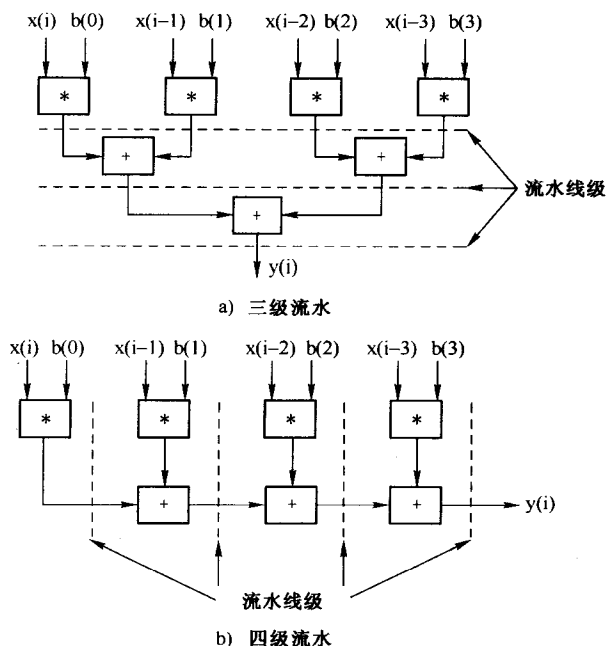


图 2-21 FIR 滤波器的两种不同数据通路

其中 N 等于 4。应该注意图 2-21a 中数据通路并发地实现了所有的乘法操作,并且通过求和树将乘积并行相加。图 2-21b 中的数据通路同样并发实现乘法操作,所不同的是之后对乘积项顺序相加。此外,注意图 2-21a 中的数据通路有 3 级流水,每步都用虚线标出;而图 2-21b 中的数据通路有 4 级相似的流水。虽然这两个数据通路都使用 4 个乘法器和 3 个加法器,但是图 2-21b 中数据通路更规则,而且用 ASIC 技术更容易实现。

在这种体系结构中,只要算法中的每个操作都通过本身的单元实现(如图 2-21 中所示),就不需要系统控制,因为数据只是简单地从一个单元流向下一个单元,并且时钟用来加载流水寄存器。然而,有些时候,需要减少单元的使用量以便节省硅片面积。在这种情况下,需要有简单的控制器来控制数据在单元和寄存器间的传输,并且要为那些在不同时间执行不同函数的单元来选择合适的代数函数。另一种情况是,对于同一个数据通路可能实现多于一个不同算法,并且每个算法在不同时间执行。在这种情况下,因为每个算法都需要一个独特的数据流来流过数据通路,所以我们需要控制器来管理数据流。这样的控制器通常很简单而且没有条件分支。

一般来说,数据通路体系结构常用 ASIC 实现。一个代价较少的解决方案是使用出售的 DSP 处理器,其数据通路由一个乘法器和一个累加器构成。这种 DSP 处理器通常有一个模拟接口,在一个芯片上带有 A/D 和 D/A 转换器。同样,它通常对于指令和数据有各自的总线,所以读取指令和访问数据可以并发进行,因此可以得到较高的吞吐率。在某些情况下,它也许有预先编程好的 ROM 来执行标准化函数,比如说傅里叶变换和数字滤波。

2.9.3 带数据通路的有限状态机

带数据通路的有限状态机(finite-state machine with datapath ,FSMD)是一种综合了 FSM 控制器和数据通路的体系结构,正如图 2-22 中的方框图所示。值得注意控制单元包含一个状态寄存器和两个组合模块,分别表示次态函数和输出函数。数据通路本身包含的功能单元有 ALU、乘法器和移位器、存储组件(像寄存器和存储器)等,以及用于连接这些不同的组件的选择器和总线。控制单元的输入是数据通路的状态信号,而输出是用于控制数据通路中的功能或存储组件的控制信号。数据通路的输入和输出都连接至一个或多个存储器,提供数据通路中计算所用的数据以及存储结果数据。

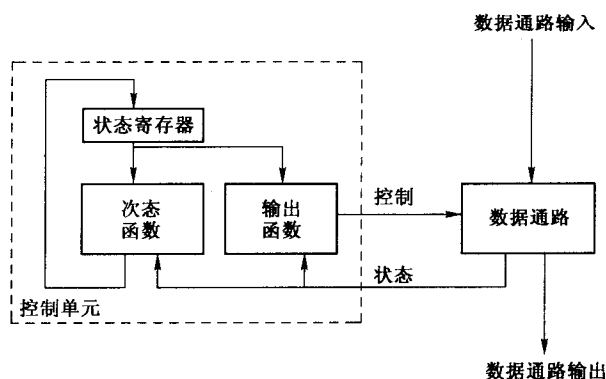


图 2-22 通用 FSMD 框图

作为一个通用体系结构,FSMD 用于不同的 ASIC 设计。上述 FSM 控制器和 DSP 数据通路是这种体系结构的两个特例。除此之外,FSMD 还是通用处理器的基本体系结构,因为每个处理器除了数据和程序存储器之外,还包括一个控制单元和数据通路。

2.10 处理器

2.10.1 复杂指令集计算机

开发复杂指令集计算机(complex-instruction-set computer,CISC)的体系结构最主要的目的是为了减少编译代码中指令的数目,使取指令操作所需要的内存访问数量达到最小化。在过去这个目的很有根据,因为内存价格贵而且比处理器速度慢。开发 CISC 的第二个目的是简化编译器结构,它在处理器指令集中包含了类似于程序设计语言结构的复杂指令。这些复杂指令减少了程序设计语言和机器语言之间的语义差别,而且简化了编译器的构造。

为了支持复杂指令集,CISC 通常有一个同样复杂的数据通路,以及一个微程序控制器,如图 2-23 所示。该控制器又由一个微程序内存、一个微程序计数器(MicroPC)和地址选择逻辑构成。在微程序内存中的每一个字都表示一个控制字,并且包含了一个时钟周期内所有数据通路控制信号的值。这就意味着控制字中的每一位表示一个数据通路控制线的值,例如它可以用于加载寄存器或是选择 ALU 中的一个操作。此外,每个处理器指令都由一系列的控制字组成。当从内存中取出这样的一条指令时,首先把它放在指令寄存器中,然后地址选择逻辑再根据它来确定微程序内存中相应的控制字顺序的起始地址。当把该起始地址放入 MicroPC 中后,就从微程序内存中找到相应的控制字,并利用它在数据通路中把数据从一个寄存器传送

到另一个寄存器。因为 MicroPC 中的地址并发递增来指向下一个控制字,因此对于序列中的每个控制字都会重复一遍这一步骤。最终,当执行完最后一个控制字时,就从内存中取出一条新的指令,整个过程将会重复进行。

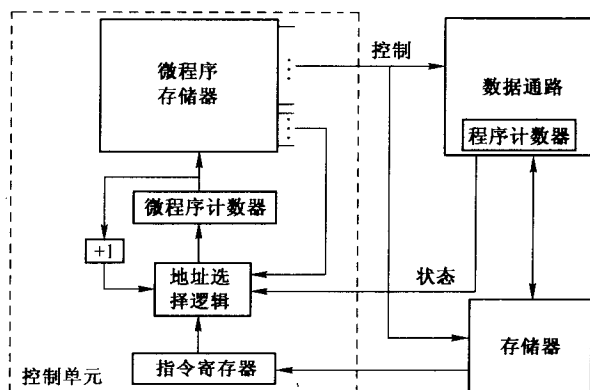


图 2-23 微程序控制的 CISC

52

由此可见,控制字的数量以及时钟周期的数目对于每一指令都可以是不同的。因此,在 CISC 中很难实现指令流水操作。另外,速度相对较慢的微程序存储器需要一个比必要时间要长的时钟周期。因为指令流水和短的时钟周期都是快速执行程序的必要条件,因此 CISC 体系结构对于高效处理器来说是不太适合的。

虽然 CISC 体系结构可以执行不同的复杂指令,但是程序执行的统计数据表明:最常用的指令都很简单,而且只含有少量几种寻址方式和数据类型。统计数据同时也表明,大部分的复杂指令很少或者根本不会被使用。导致复杂指令使用较少的原因是程序设计语言结构和所使用的复杂指令之间存在微小的语义差别,以及将程序结构映射到这样的复杂指令时所面临的问题。由于这个问题的存在,复杂指令在 CISC 体系结构的优化编译器中很少被使用,因此也降低了 CISC 体系结构的实用性。

内存价格的稳定下降和速度的提高使得程序代码紧密优化和复杂指令集对于高效计算来说是不必要的。此外,复杂指令集使得 CISC 体系结构的优化编译器的构造成本太高。基于这两个原因,CISC 体系结构被下一节将要阐述的 RISC 体系结构所替代。

2.10.2 精简指令集计算机

相对于 CISC 体系结构,精简指令集计算机(reduced-instruction-set computer, RISC)体系结构经优化可以达到较短的时钟周期,每个指令所需周期较少,并有有效的指令流水操作。如图 2-24 所示,RISC 处理器的数据通路通常由一个大的寄存器文件和一个 ALU 组成。一个大的寄存器文件是十分必要的,因为它包含了程序计算中所有的操作数和结果。通过 load 指令将数据放到寄存器文件,通过 store 指令将其放回内存。这个寄存器文件越大,代码中 load 和 store 指令的数目就越少。当 RISC 执行一条指令时,指令管道首先将指令放到指令寄存器中。然后将该指令解码并从寄存器文件中读取合适的操作数。第三步,RISC 做下面两件事情之一:或者在 ALU 中执行所需的操作,或者从数据缓存里面读/写数据。应注意每个指令的执行仅仅占用大约三个时钟周期,这就意味着指令流水线可以很短小且高效,而仅在数据或者

53

分支相关性的情况下才会使用较多周期。

我们同时注意到,因为所有的操作数都包含在寄存器文件里面,而且只使用了几种简单的寻址方式,所以同样可以简化数据通路的设计。此外,因为每个操作要执行一个时钟周期而每个指令要执行三个时钟周期,因此控制单元也可以很简单,而且可以使用随机逻辑而不是微程序控制来实现。总之,RISC中控制和数据通路的简化导致了简短的时钟周期,并最终达到了更高的性能。

然而,应该指出的是,RISC体系结构的极大简化则要求一个更为复杂的编译器。

54

比如说,RISC设计不会在指令相关性发生

时就停止指令流水线,这就意味着编译器有责任产生出无相关性的代码,或者可以通过时延指令的产生,或者对指令进行重新排序。而且由于指令数目的减少这一事实,RISC编译器将会需要一系列的RISC指令来完成复杂的操作。当然,虽然这些特点需要更为复杂的编译器,但是它们同时也给予编译器极大的灵活性来实现积极的优化。

最后,我们应该注意到由于未采用复杂指令,RISC程序往往需要多出20%到30%的程序内存。然而,因为较为精简的指令集可以使得编译器设计和运行时间更为简短,所以最终的编译后的代码效率是很高的。此外,由于这些较为简单的指令集,RISC处理器往往比那些CISC处理器节省硅片面积,并缩短了设计周期。

2.10.3 向量机

向量机(vector machine)发展于20世纪70年代早期,是为了满足不断增长的对于高性能计算的需求,这需要减少时钟到最小值才可以达到。因为时钟周期大约是内存访问或算术运算所需时间的十分之一,所以设计者们过去常常用许多的流水步来实现内存和算术单元。在这样的一个向量机中,内存由许多并行模块组成,而所有模块中的数据都根据同一地址并发存取。简而言之,数据的一个向量是并发存取的。每个向量都从向量寄存器中通过内存存取流水线(内存管道)串行移入或移出。每个向量指令都对两个向量操作数操作,并且生成一个向量结果。正如图2-25所示,向量机同时也包括标量寄存器以及执行标量运算的标量功能单元。

向量机的主要优点是只要代码中有足够的向量操作,则其速度就非常快。然而我们必

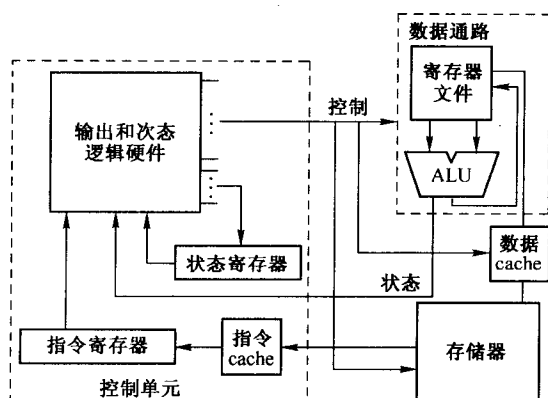


图 2-24 硬件控制的 RISC

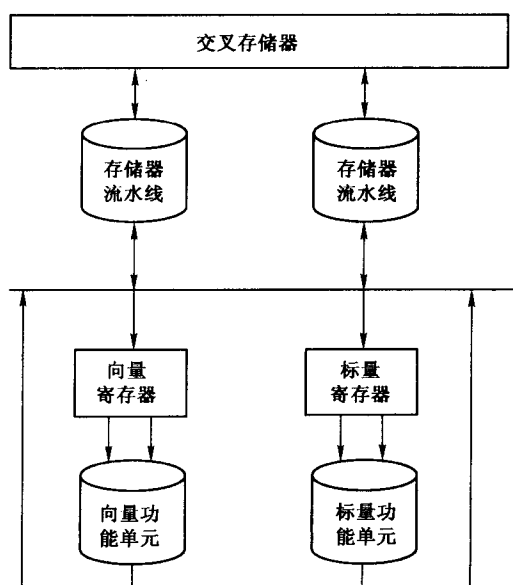


图 2-25 向量机

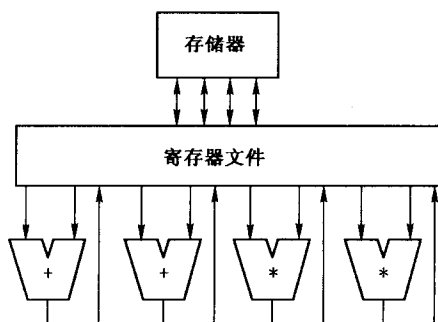
须注意到,普通的代码并不是特意为了向量机所编写的,这就意味着向量机需要一个向量化编译器来优化普通代码以适合向量指令。通常来说,这样的向量化工作是通过将一个循环体分解成尽可能多的向量操作,而其中每一个向量操作都可以看做只含有单一操作的循环体。此外,我们也应该看到在处理标量操作和有着非线性标引的数组存取时,向量机的性能往往会有所下降。

55

2.10.4 超长指令字计算机

超长指令字计算机(very-long-instruction-word computer, VLIW)通过在数据通路中使用多个功能单元来体现并发性,并且所有的功能单元都在一个中心控制下按照一个锁步骤的机制来执行。一个 VLIW 指令对于每个功能单元都有一个对应的域,并且 VLIW 指令中的每个域都标明了源和目的操作数的地址以及该功能单元要执行的操作。因此,一个 VLIW 指令通常很长,因为它的每个功能单元都几乎包含了一条标准的指令。

在图 2-26 中,我们可以看到一个 VLIW 数据通路的实例,它由四个功能单元组成:两个 ALU 和两个乘法器,一个寄存器文件和一个存储器。为了利用所有这四个功能单元,这个例子中寄存器文件有 16 个端口:8 个输出端口,用于为功能单元提供操作数;4 个输入端口,用于存储从功能单元得到的结果;还有 4 个输入/输出端口,用于存储器间的通信。现在有一个很有趣的现象值得注意:假设在 VLIW 上执行的代码有四个并行路径,这就使得 VLIW 在每个时钟周期内可以执行四个独立的指令,我们可能



56

图 2-26 VLIW 数据通路实例

想像图 2-26 中的 VLIW 所提供的性能应该是只含有单一功能单元的处理器性能的 4 倍。但是在实际中,大部分的代码都存在并行部分和串行代码相互交叉的情况。因此,带有大量功能单元的 VLIW 很有可能不会被完全利用。我们仍然需要假定理想的条件是所有的操作数都存放在寄存器文件中,而每个时钟周期要从中取出 8 个操作数并存入 4 个结果,此外还要从存储器中取出 4 个新的操作数来作为下个时钟周期之用。但是应该注意到,因为某些结果必须存回给存储器以及某些结果在下个时钟周期中并不需要,因此这一计算模式并不容易达到。在这些条件之下,VLIW 数据通路的效率就没有理想状态下那么高。

最后,我们应该指出存在着两个技术局限性,影响 VLIW 体系结构的实现。第一,当我们构建了一个 8 到 16 个端口的寄存器文件时,那么一旦我们超出了这个数目限制时,这些寄存器文件的效率和性能往往会快速下降。第二,因为 VLIW 的程序和数据存储器都需要较高的通信带宽,因此这些系统往往也要求昂贵的高端打包技术。总之,这就是为什么 VLIW 体系结构不如 RISC 体系结构受欢迎的原因。

57

2.11 并行处理器

在并行处理器的设计之中,我们可以通过使用并行工作的多个处理单元(PE)来利用它们的并行性。在这种类型的体系结构中,每个处理单元可能包括自己的带有寄存器和本地存储器的数据通路。并行处理的两种典型的类型分别是 SIMD 处理器(单指令流多数据流)和

MIMD 处理器(多指令流多数据流)。

在 SIMD 处理器(通常叫做阵列处理器)中,所有的处理单元按照一个锁步的机制来执行同一条指令。为了将这条指令传播到所有处理单元以及控制它们的执行,我们通常使用一个单一的全局控制器。通常情况下,阵列处理器都是绑定在一个主处理器上,这就意味着它可以被看成是一种对于需要集中计算的任务的硬件加速器。这种情况下,主处理器会把数据加载到各个处理单元中,而后当计算完成后再统计结果。而且如果需要,处理单元还可以直接和它们最近的邻居进行通信。

阵列处理器的最大的优点就是非常适合于那些可以自然映射到矩形网格上的计算,比如图像处理操作,因为一个图像可以分解成矩形网格里面的像素点;再比如天气预报,因为地球的表面可以分成 $n \times n$ 英里^①的方格。因为所有的处理单元都执行相同的指令流,所以在矩形阵列处理器中的网格点上编写程序变得很简单。然而,对于将要在阵列中的数据路由选择进行程序设计就变得很困难,因为要求程序员不得不对每个时钟周期中每个数据的所有位置都了如指掌。基于这个原因,像矩阵的三角化或是转置操作这些问题在一个阵列处理器上很难编程进行处理。

只有当问题的自然结构和阵列处理器的拓扑结构很匹配时,阵列处理器才是易于构建并且是易于编程的。因此,由于用户很难对普通的问题类型编写程序,所以不应该把它们当作通用计算机。

MIMD 处理器,通常叫做多处理器系统(multiprocessor system),它和 SIMD 的不同之处在于它的每个处理单元都执行自己的指令流。在这种类型的体系结构中,问题既可以由主处理器进行加载,也可以由每个处理器从共享内存中加载自己的问题。通过采用图 2-27 中所示的一种或者两种通信机制,每个处理器可以和这个多处理器系统中的其他任何处理器进行通信。在共享内存多处理器系统(shared-memory multiprocessor)中,所有的处理器都通过 N 端口 $\times N$ 端口的互联网络连接到一个共享内存上,这就意味着每个处理器都可以对共享内存中的任意数据进行存取。而另一方面,在消息传递多处理器系统(message-passing multiprocessor)中,每个处理器都往往拥有自己的一块大的本地内存,并且以消息包的形式通过互联网络向其他处理器发送数据。共享内存

使用的互联网络必须要求速度快,因为它常常用于小规模数据的通信,比如一个单字。相反地,消息传递使用的互联网络往往速度要慢一些,因为它的使用频率不高而且用于通信包括许多个数据字的大消息。由于速度要求的不同,设计一个共享内存使用的互联网络要比设计一个消息传递使用的互联网络要困难。最后,应该注意到多处理器系统更加易于编程,因为它们

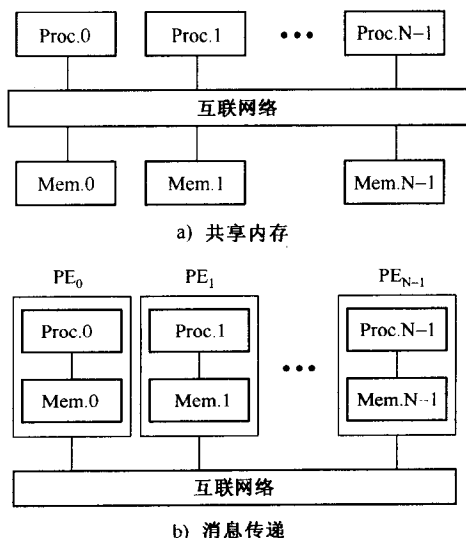


图 2-27 多处理器通信机制

① 1 英里 = 1 609.344 米。

不是面向指令的,而是面向任务的。每个任务都独立执行,而且如果有必要,在任务完成之后可以进行同步化。因此,多处理器使得程序的编写、数据的划分、代码的并行化和编译工作都要比阵列处理器更为简单。

2.12 结论

本章讨论了用于表示系统的不同种类的模型以及用于实现系统的不同种类的体系结构。不论是模型还是体系结构对于设计过程都是至关重要的,因为这一过程的整体目标就是通过体系结构中提供的一组物理构件来实现模型中表示的所要求的系统功能。更为重要的是,本章还阐明不同模型强调系统的不同方面——其状态、活动、数据或是控制。有了这个可选择的关注点,设计者选择一个模型适于表示所设计的系统的最为重要的特点,是至关重要的。而且,因为评测一个好的语言是根据它直接表示所求模型的程度,所以设计者选择适合于这一模型的语言也是同等重要的。然而有趣的是,系统描述最常用的语言通常并不能直接描述给定系统的最好的模型。解决这一问题将是以下三章的话题,我们到时将会考虑其中的原因、结果以及解决方法。

2.13 练习

1. 我们可以使用不同模型将一个系统概念化,比如说状态机、流程图、算法等。讨论对于下列系统最适合的概念模型:
 - (a) 一个模 10 计算器;
 - (b) 电话应答机;
 - (c) 一个寻找一组数字的中值的算法;
 - (d) 温度控制器;
 - (e) 存取内存的通信协议;
 - (f) 存取微处理器的通信协议。
2. 层次化的 FSM 相对于过去平面的 FSM 的优点是什么?
3. 面向对象模型与面向数据、面向活动模型的区别是什么?
4. 列出流程图与结构图的所有不同之处。并用流程图来表示图 2-16 给出的结构图。
5. 列出 RISC 和 CISC 体系结构的优缺点。
6. 定义实现各个概念模型的最有效的体系结构。
7. 详细说明作为扩展 FSM 体系结构的子集的各个体系结构。
8. 说出通用和专用体系结构的不同之处。
9. 操作流水线、数据通路流水线、指令流水线和控制流水线的区别是什么?
10. 列出各个体系结构的度量的技术缺陷。
11. 在设计和实现互连网络时的必要事项是什么?

第3章 系统描述语言

在前面的章节中,我们已经了解了如何用不同的概念模型来理解、组织和定义一个系统的功能性。然而,模型仅仅是理论上的概念,我们需要使用系统描述语言,以一种具体的形式来捕获这些模型。本章将阐明不同模型之间的一些共同特性,并考虑如何利用更为一般化的描述语言来捕获这些特性。此外,我们还将特别关注那些适合于嵌入式系统设计的模型和语言。

3.1 引言

一个系统可以在任何一个不同的抽象级别进行描述,而每一级别都有其不同目的。例如,在逻辑级(logic level)描述的系统中,设计者可利用原理图输入工具捕获系统的结构。作为另一种选择,在系统组件级(system-component level)中,设计者可以不考虑任何结构细节的描述,而使用硬件描述语言(hardware description language, HDL)来描述系统中每一组件的功能性。这样一个系统组件的描述可以代表定制的硬件、内存或是执行一定指令集的处理器。最后,在概念级(conceptual level)的描述中,能够不涉及任何系统组件而表述一个完整的系统功能性。从传统方法上来说,我们希望能使用某种自然语言(如英语),在概念级描述一个新产品的功能性。然而,随着当今系统变得越来越复杂,它们常常要求采用一种新的方法完成系统的概念化设计;更进一步地说,设计者需要以一种可执行系统描述语言(executable specification language)的形式描述系统的概念。这种可执行的语言以机器可读并能模拟的方式捕获系统的功能性。

63

这种方法有以下几个优点。首先,模拟一个可执行的描述允许设计者验证系统预期功能性的正确性。在传统的方法中,设计是从自然语言描述开始的,而验证只有等到设计被完全实现为一个可模拟的系统描述后才能进行(一般来说已经是门级的原理图)。其次,系统描述能作为综合工具的输入。通过利用综合工具,可得到一个系统的实现,这样极大地减少设计的周期。再次,这样一种描述可以作为一个全面的系统文档,为设计者提供明确的系统预期功能性说明。最后,可执行系统描述可作为一个很好的中间媒体,利用它可在不同的用户和工具之间完成设计信息的交换。因为这种方法强调的是已定义好的系统组件,而这些组件通常是由不同设计者独立完成的;结果使一些与系统集成相关的问题最小化。

在第2章,为了能从概念上描述不同类型的系统,我们介绍了一些常用的模型。相对于这些概念模型,很显然任何一种描述语言的目标都是希望设计者可以通过最小的努力获取系统的概念。例如,当我们想要定义软件中面向对象的概念模型时,C++程序设计语言可能是最为行之有效的语言。但是,如果我们的目标是描述一个层次化/并发性的有限状态机模型时,Statecharts语言则可能是更为实用的选择[Har87]。

因为不同概念模型具有不同的特性,任何一种给定的描述语言都能或多或少地适合于某种模型,适合的程度依赖于它是否能支持这个模型中所有的或部分的特性。为了找到某种能直接捕获给定的概念模型的语言,我们需要在概念模型和语言结构的特性中,建立一对一的相关性。

64

在本章,我们将从描述概念设计模型中的一些共有特性开始。然后,介绍一种特殊类型系统,即嵌入式系统(embedded system)中的特性。其中,我们将分析一些普遍使用的硬件描述语言,比如 VHDL、Verilog、HardwareC、通信顺序进程(communicating sequential process, CSP)、Statecharts、规范与描述语言(specification and description language, SDL)、Silage 以及 Esterel 等,并着重讨论这些语言如何支持嵌入式系统的特性。最后,我们将介绍 SpecCharts 语言,它是一种基于程序状态机模型(在第2章已做过介绍)的语言;同时还将展示这种语言如何支持对嵌入式系统的描述。

3.2 概念模型的特性

在本节中,我们将给出概念模型中通常被设计者使用的一些共有特性。为了表述这些特性,我们的目标是评价每一个特性在描述一种或多种系统行为概念方面是如何使用的。

3.2.1 并发性

任何一个系统都可被分解成一些功能块,我们称之为行为(behavior)。而每一个这样的行为都可以几种方式来表示,例如一个进程、一个过程或是一个状态机。在多数情况下,一个系统的功能性最容易被表示成一些并发行为的集合,因为只简单地使用顺序结构来定义系统功能性可能会导致一个难于理解并十分复杂的描述。然而,如果我们能找到一种可表示并发性的方式,那么对于这样的系统,我们通常能得到一个更为自然的描述。例如,考虑一个仅有两个并发行为的系统,它们通过有限状态机 F_1 和 F_2 分别描述。系统的一个顺序表示将可能是两个有限状态机叉积的结果,即 $F_1 \times F_2$,势必产生一个巨大数目的状态集。而一个更为优雅的方法是,使用由两个或更多并发有限状态机组成的概念性模型来定义其系统的行为,类似于 Statecharts[Har87]模型和 PSM[GVN93]模型。

并行性的概念可应用到任何一个不同的抽象级别中[HB85]。例如,在作业级(job-level)并发中,其并行性可以表示为利用多重编程(multiprogramming)、多重处理(multiprocessing)和时间共享(time sharing)机制,来实现几个作业间的并行执行。相对而言,在任务级(task-level)中,并发则是指组成一个作业的不同任务同时执行。许多系统都是在这一级别中描述,系统中的每个任务通过行为来表示。语句级(statement-level)的并发指的是在一个任务中不同语句间的并行执行。语句级并发的一个实例可以是在向量处理器上一个循环体内语句的执行。除此之外,还有操作级(operation-level)的并发,它指定一个系统中不同操作的并发执行。例如,一个加法操作可以和一个乘法操作并发执行。操作级的并发通常在处理器、过滤器以及数字信号处理器中都能找到。最后,并发也存在于位级(bit-level)粒度中,比如在一个算术逻辑运算器中逐位计算的情况。

并行性的表示可分为两大类,数据驱动(data-driven)或控制驱动(control-driven)。这依赖于在行为中动作的执行顺序。

1. 数据驱动的并行性

一些行为可被显式地描述为操作或语句的集合,而没有指定其显式的执行次序。在这种情况下,执行的次序可以仅由它们之间的数据依赖性决定。换句话说,每一个操作将针对于其输入的数据执行计算,然后输出新的数据;而这些数据接着又被作为其他操作的输入。在这样的数据流描述中,操作的执行仅依赖于数据的可用性,而并不依赖于操作或语句在系统描述中

的物理位置(physical location)。数据流表示以单一赋值规则(single assignment rule)为特性,这意味着每一个变量只能在赋值语句的左边出现一次。

例如,考虑如图 3-1a 中所示的赋值语句。和其他数据驱动的执行不一样,该数据流中对 p 的赋值却在使用 p 值计算 y 值的语句之后,这种情况是会产生任何结果的。不管语句的执行次序,这些操作的执行仅由数据的可用性决定,如图 3-1b 所示的数据流图。通过遵循这个规则,我们可看到,因为 a 、 b 、 c 、 d 是原始的数据输入,所以语句 1 和语句 3 中的加法和减法操作首先被执行。接着,这两个计算的结果将会作为语句 3 中乘法所需的数据。最后,在语句 2 中的加法被执行,计算 y 值。

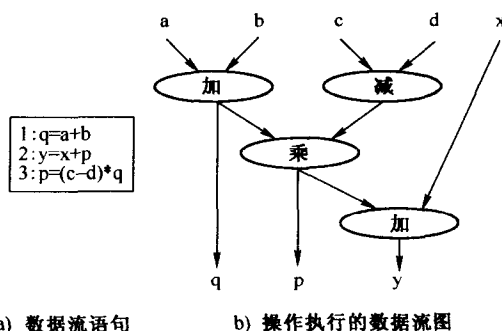


图 3-1 数据驱动的并发性

数字信号处理系统是一个可以说明数据驱动并发性的例子,它对连续的数据采样流进行操作。然而,应当注意的是:像这样的系统可能要求额外的存储设备,以防数据流中一些采样数据将被后续的计算使用。

2. 控制驱动的并发性

控制驱动并发性中的关键概念是控制线程(control thread),它被定义为系统中必须顺序执行的操作集合。正如前面介绍数据驱动并发时所说,操作之间的依赖性决定其执行顺序。相对而言,在控制驱动的并发中,它是由控制线程或多线程来决定执行顺序的。换句话说,控制驱动的并发性的特点是通过使用指定多个控制线程的显式结构,而所有这些线程都并行执行。

任务级和语句级都可指定控制驱动的并发性。在任务级中,如分叉-结合结构(fork-join)和进程结构(process)通常可以用来指定操作的并发执行。更为详细地说, **fork** 语句创建一组并发控制线程;而 **join** 语句则等待前面分叉的各并发控制线程完成。例如,在图 3-2a 中 **fork** 语句,产生了 3 个控制线程 A、B 和 C,它们都以并发的方式执行。与此相关的 **join** 语句必须等待所有的 3 个线程都结束后,其后的 R 函数中语句才能相继执行。在图 3-2b 的进程语句中,我们知道进程语句是如何指定并发性的。应当注意:一个 **fork-join** 语句结构是从一个控制线程开始,分叉成几个并发线程,如图 3-2c 所示;而进程语句 **process** 则是把行为表示成并发线程的集合,如图 3-2d 所示。例如,在图 3-2b 的进程语句 **process** 中,创建了 3 个进程 A、B、C,每一个进程代表一个不同的控制线程。分叉-结合和进程这两

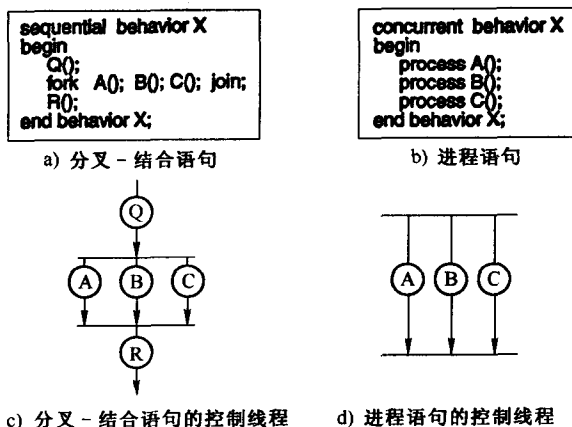


图 3-2 控制驱动的并发性

中,创建了 3 个进程 A、B、C,每一个进程代表一个不同的控制线程。分叉-结合和进程这两

68 种语句结构可以相互嵌套,分叉-结合语句结构还能使用嵌套的进程语句来实现,也可以反过来实现。从这种意义上来说,这两种方法是相互等价的。

在语句级中,可用两种方式之一指定并发性。在显式指定的语句级并发中,利用特殊的结构指定几个语句并行执行。例如,考虑 HardwareC[KD88]中的并行复合语句(parallel compound statement):

```
<
  x = b + c;
  y = p - q;
>
```

上面的语句指定对 x 和 y 的赋值计算同时执行。

语句级的并发同样能以隐式的方式指定。例如,在未来的某个时间里安排更新一些值,可能会使用隐式的语句级并发。如下面例子中的语句:

```
s <= b + c after 20ns;
wait for 20ns;
v := 3;
```

这些语句指定信号 s 的值以及变量 v 的值都将在 20 纳秒后同时得到更新。

3.2.2 状态迁移

与控制器、电信系统的运行模式相类似,一些系统通常被定义为具有不同的行为模式或状态。例如一个交通灯控制器[DH89],在白天和夜晚的操作可能会有不同的模式,既可以是手动的又可以是自动的;也可以根据交通灯自身的状态来改变其模式。

在具有不同模式的系统中,这些模式之间的迁移有时是以一种无结构的方式发生,这和模式的线性顺序(linear sequencing)相反。这种任意的迁移类似于在程序设计语言中使用 goto 语句。如图 3-3 所示,描述了一个在模式 P 、 Q 、 R 、 S 和 T 之间相互迁移的系统,它们的迁移次序仅仅是由某种条件决定的。假定一个有 N 个状态的状态机,那么在那些状态之间存在 N^N 种可能的迁移。

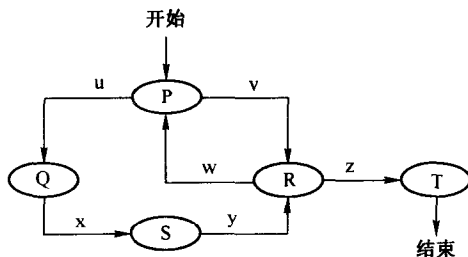


图 3-3 任意复杂行为之间的状态迁移

在类似这样的系统中,不同模式下的迁移可通过检测某种事件或条件的发生来触发。例如,在图 3-3 中,当事件 u 发生后,状态 P 将迁移到状态 Q 。在一些系统中,每一次迁移还可能伴随着动作的发生,而在每一个特殊模式或状态下可能有任意复杂的行为或计算。在交通灯控制器例子中,在某一个状态下可能仅是红、黄、绿灯之间的顺序转移;然而在另一个状态下,它可能需执行一个算法来决定哪条路具有更高的优先权。这种算法可以以白天的时间段和交通的密度为基础。在传统的层次化有限状态机概念模型中,简单的赋值语句,如 $x = y + 1$,可与一个状态相关联。在 PSM 模型[GVN93]中,任意具有迭代和分支结构的算法可和一个状态相关联。

3.2.3 层次化

对于大型系统而言,我们遇到的问题之一是它们过于复杂,难以用整体的方式来进行考虑。在这样的情况下,我们希望利用层次化模型的优点。首先,因为层次化模型允许将一个系统定义为一些更小子系统的集合。这样,系统建模者可在一段时间内集中精力对某一子系统进行建模。系统中这种模块分解的方法可简化系统概念上的开发。而且,我们一旦进入了适当的概念阶段后,这种层次化模型能极大地帮助我们理解系统的功能性。最后,一个层次化模型为限制对象作用域提供了一种机制,这些对象可以是所声明的类型、变量和子程序名称等。如果没有层次化的概念,可能会使所有的对象全局化,这样难以将它们在模型中的特殊用途与之相关联,同时也使我们难以在同一个模型中不同部分重用这些命名。

有两种不同类型的层次化:结构的层次化和行为的层次化。它们通常都能在概念模型的系统

1. 结构层次化

结构层次化是将系统的描述表示成互联元件的集合。每一个元件都有其自己的内部结构,这些结构又是由一个更低层次的互联元件集合组成,以此类推,可以分解为更小的互联元件。在元件间的互联表示连接元件的线网集合。这样一种模型可表示一个结构的层次化,其优点在于它能帮助设计者用一个已存在的元件集合来定义新的元件。

结构层次化可在系统的不同抽象级别中描述出来。例如,一个系统可以分解为芯片/模块的集合,它们之间通过总线相互通信。一个芯片可以由几个块(block)组成,而一个块又可以由几个寄存器传输(RT)元件构成,如寄存器、算术逻辑运算单元(ALU)和多路器等。最后,一个寄存器传输元件进一步被分解成门的集合。此外,我们需注意的是系统的不同部分能在不同的抽象级别中描述,如图 3-4 所示,图中处理器已经被结构化地分解为由一系列 RT 元件组成的数据通路和由门集合表示的相关控制逻辑。

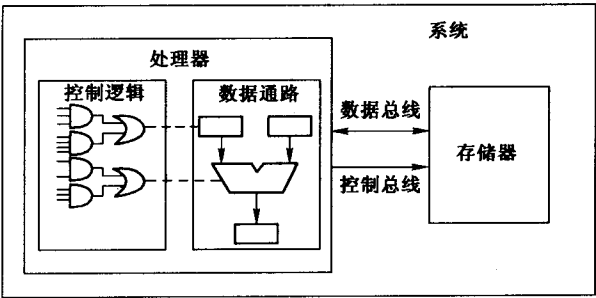


图 3-4 计算机系统结构层次化

2. 行为层次化

行为层次化的描述是一个将行为分解成若干不同子行为的过程,这些子行为可以是顺序执行的关系,也可以是并发的关系。

一个行为的顺序分解(sequential decomposition)可以表示为一些过程集合或是一个状态机。在第一种情况下,行为的过程顺序分解(procedural sequential decomposition)是把行为定义为过程调用序列。即使是由一个顺序语句集合组成的行为,我们也可以看成是把这些语句封装在一起的一个过程。图 3-5a 显示了行为 P 的过程顺序分解,行为 P 由顺序执行的子行为

70

71

组成,分别用过程 Q 和 R 表示。图中行为的层次化可以通过嵌套的过程调用描述。过程中的递归调用允许指定动态行为层次化,这意味着层次化的深度将在运行时决定。

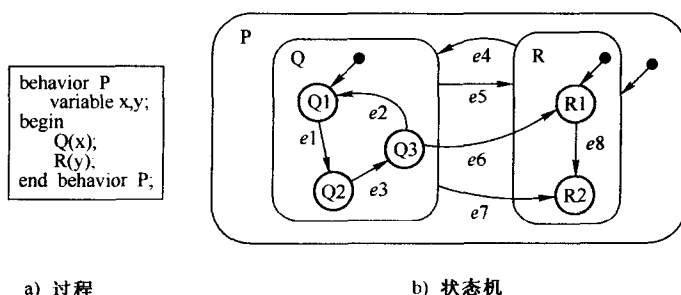


图 3-5 顺序行为分解

图 3-5b 给出了行为 P 的状态机顺序分解 (state-machine sequential decomposition)。在图中, P 分解为两个顺序子行为 Q 和 R , 每一个子行为表示状态机中的一个状态。通过允许将一个子行为表示成另一个状态机, 这种状态机的表示方式可体现出其层次化。因此, Q 和 R 是状态机, 所以它们可进一步分解成顺序子行为。在最低层的行为中, 包括 $Q1, \dots, R2$, 它们被称为叶子行为 (leaf behavior)。

在一个顺序分解的行为中, 通过几种类型的迁移: 简单迁移、成组迁移和层次化迁移, 表示子行为之间的关联。一个简单迁移 (simple transition) 类似于在一个 FSM 模型中相互连接的状态, 它能引起在行为层次中同一级别的两个状态间的迁移。如图 3-5b 所示, 由事件 $e1$ 触发的迁移使控制权从行为 $Q1$ 传递给行为 $Q2$ 。成组迁移 (group transition) 指的是一组状态的迁移, 比如事件 $e5$ 引发从 Q 的任何子行为向行为 R 迁移的情况发生。层次化迁移 (hierarchical transition) 则是跨越了几个行为层次级别的迁移, 同时它也包括简单迁移和成组迁移。例如, 由标记为 $e6$ 的迁移将控制权从行为 $Q3$ 传递给行为 $R1$, 这意味着它必须跨越两个层次级别。同理, 标记为 $e7$ 的迁移将控制权从 Q 传递给较低层次的状态 $R2$ 。

对于一个顺序分解的行为, 必须显式指定其初始的子行为, 这个子行为应当在该行为被激活的时候执行。如图 3-5b 中, 当 R 的父行为 P 被激活时, R 是第一个执行的子行为, 因为有一个用带圆点箭头从父行为指向 R 。同理, $Q1$ 和 $R1$ 分别是行为 Q 和 R 的初始子行为。

行为的并发分解 (concurrent decomposition) 可通过分叉 - 结合结构或是利用进程集合建模系统的方式来表示。这些结构已经在 3.2.1 节中讨论。

3.2.4 程序结构

许多行为可很好地描述成一些顺序的算法。例如, 考虑对一个数组中存储的数字集合进行排序的系统, 或是一个用来生成随机数集合的系统。在这样的情况下, 如果系统设计者设法把行为层次化分解为越来越小的子行为, 将最终使每个子行为的功能性都能用一个算法直接实现。

使用程序结构 (programming construct) 指定行为的优点在于: 允许系统建模者对系统中的计算行为指定其显式的执行顺序。虽然可以利用一些注释来描述算法, 但程序设计语言结构仍然被相当广泛地使用。这些结构包括赋值语句、分支语句 (if/case 语句)、循环语句 (while、for 和 repeat 循环语句) 以及子程序调用 (function 函数和 procedure 过程)。另外,

数据类型如记录类型、数组以及链表通常对复杂的数据结构建模都非常有用。

图 3-6 显示了我们怎样使用程序设计结构指明一个以降序方式排序 10 个整数集合的行为。注意过程 *Swap* 交换它的两个参数值。

3.2.5 行为完成

行为完成 (behavioral completion) 指的是一种表明自身行为已经完成的能力, 同时也是一种能使其他行为检测到这个完成状态的能力。当一个行为中所有的计算都已执行完毕, 并且所有必须被更新的变量都已经用新值写入后, 这个行为才能称之为完成。

在有限状态机模型中, 我们通常指定一个显式定义的状态集合为最终状态 (final state)。这意味着: 对于一个状态机而言, 完成将会在控制流转移到这些最终状态之一时发生, 如图 3-7a 所示。

```
type    buffer_type is array (1 to 10) of integer;
variable buf : buffer_type;
variable i, j : integer;

for i = 1 to 10
  for j = i to 10
    if (buf(i) > buf(j)) then
      Swap(buf(i), buf(j));
    end if;
  end for;
end for;
```

图 3-6 使用程序设计结构表示的排序行为

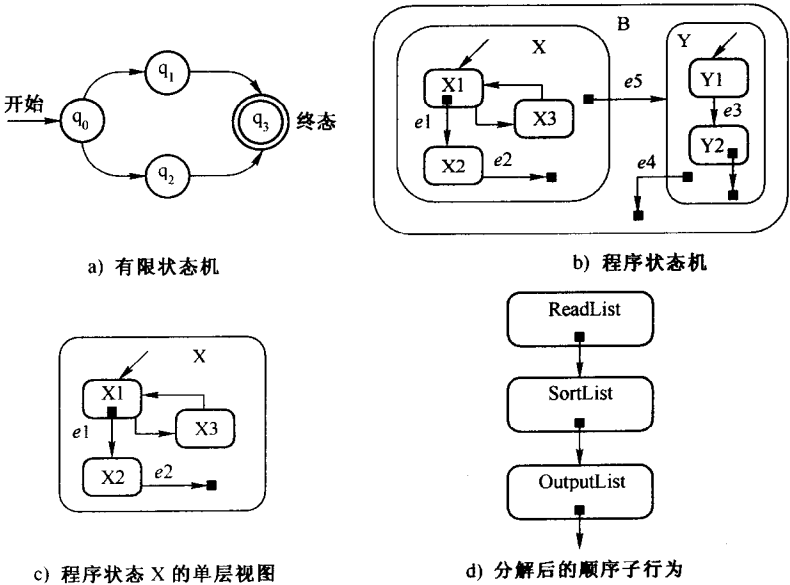


图 3-7 行为完成

在我们使用程序设计语言结构时, 当程序中的最后一个语句已执行时, 我们认为一个行为将完成。例如, 当控制流转移到一个 *return* 语句时, 或是当过程中的最后一个语句执行时, 即称一个过程已完成。

在第 2 章中引入的 PSM 模型, 使用一个特殊的预定义的完成点 (completion point) 来指明其行为的完成。当控制转移到这个完成点时, 称包含该完成点的程序状态已经完成。在这个完成点上, 仅当源程序状态完成后才能到达的完成迁移弧 TOC (transition-on-completion) arc 现在已到达。

考虑图 3-7b 中的程序状态机的例子。在图中, 叶程序状态的行为 (如 X1) 用程序设计结

构描述,这意味着它们的完成是以最后一条语句的执行来规定的。对于 X , 程序状态机的完成点使用实体方形表示。当控制流从程序状态 $X2$ 转移给 X 时(例如,当到达事件 $e2$ 所标识的 TI 弧时),表示程序状态 X 已完成。接着,事件 $e5$ 将引起一个 TOC 弧迁移给程序状态 Y 。同理,当控制流经事件 $e4$ 标识的 TOC 弧,从程序状态 Y 迁移到 B 的完成点,表明程序状态 B 已完成。

这种对于行为完成的描述有两个优点。首先,在层次化的描述中,完成点能帮助设计者定义每一个层次级别,并且能把每一个层次看成是一个独立的模块,免于让设计者考虑不同层之间相互迁移的影响。例如,图 3-7c 展示了在图 3-7b 中的程序状态 X 是怎样把其自身隔离于整个大系统之外的。将 X 的行为功能性分解成三个程序子状态 $X1$ 、 $X2$ 和 $X3$,系统建模者并不需要考虑事件 $e5$ 标识的完成迁移所带来的影响。从这个观点来看,设计者可独立地开发 X 的程序状态机行为,并结合它自身的完成点(由事件 $e2$ 标识的从 $X2$ 完成的迁移)。指定行为完成的第 2 个优点在于它允许将一个行为自然地分解为一些子行为,而这些子行为之间是通过“完成”迁移弧来决定顺序的。例如,图 3-7d 显示了一个对一系列数字进行排序的应用,我们将之分解成有 3 个有意义的子行为: $ReadList$ 、 $SortList$ 和 $OutputList$ 。因为 TOC 弧能够决定这些行为顺序,所以系统并不要求额外的事件来触发它们之间的状态迁移。

3.2.6 通信

一些系统是由几个彼此通信的并发行为或进程组成的。通常,系统中不同部分之间的通信是以共享内存(shared memory)或消息传递(message passing)的方式来描述的。我们下面将分别讨论这两种通信方式。

1. 共享内存方式的通信模型

在共享内存模型中,每一个发送进程都能向一个共享媒体(shared medium)写入数据,比如说全局变量或端口,而它们能被所有的接收进程读取访问。如果在相互通信的进程之间需要同步机制,则必须显式指定。例如,一个发送进程可以通过加入一个特殊的“有效”(valid)标志,表明共享内存已经更新为新值,而后更新的新值可被接收进程读取。这种共享内存模型也含有一个广播机制(broadcast mechanism),它能确保由进程或其环境所产生的任何一个值或事件都能立刻被所有其他进程感知。

图 3-8a 表明进程 P 和进程 Q 之间是如何通过使用共享内存方式来进行通信的。为了把数据发送到进程 Q ,进程 P 更新了共享内存中的数据,接着进程 Q 读取新的数据。

通常,完成进程间通信的共享媒体可分为持久或非持久两类。持久(persistent)的共享媒体能保持一个进程写入的数据值,直到被其他进程重写。例如一个存储单元,如寄存器、锁存器或是一个内存,其他的进程可以在任何时候访问这些存储单元中所保存的值。相对而言,非持久(non-persistent)的共享媒体中,由于在两次连续写的间隔中不能在媒体中保持写入的数据,因此仅仅在其值被一

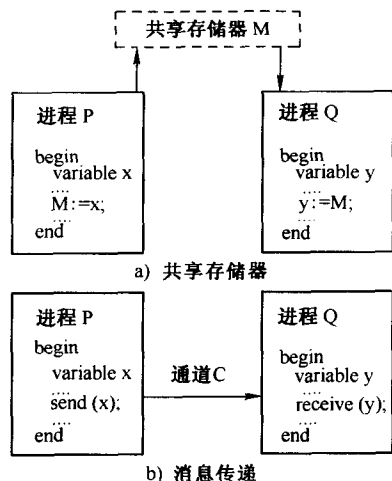


图 3-8 进程间通信方案

个进程刚写入时数据才可以利用。非持久共享存储器的例子如实现两个进程间通信的模块之间的连接线网。

77

2. 消息传递方式的通信模型

在消息传递模型中,进程间数据传递的细节都可通过在一个抽象媒体上的通信来代替,称之为通道(channel),它能传送数据或消息(message)。在每一个进程里,可使用发送-接收原语(send-receive primitive)在通道上传输数据。

考虑图 3-8b 中的例子,它显示了进程 P 和 Q 之间的通信是如何通过使用消息传递模型来实现的。通道 C 被定义为用来从进程 P 到进程 Q 传输数据的中间媒体。在进程 P 中用变量 x 表示的数据值,通过使用“发送”原语被传输到通道上。最后,通过“接收”原语将数据写入到进程 Q 中的变量 y 。

需要强调的是通道只是一个虚拟的实体,因此它可让设计者免于考虑任何与实现相关的细节。仅在综合之后,一个通道才得以实现,它由线网集合组成的总线来代替,同时在那些线网上给定了一种顺序化数据传输的协议。

78

在消息传递模型中有以下几种不同的方式。首先,在“发送”原语中显式指明目标进程的标识,或者将目标进程的标识隐式地保留在通道的互联描述中。其次,一个通信通道或者是单向的(uni-directional)或者是双向的(bi-directional),这依赖于数据是否能以单向或双向发送。最后,一个通道可以是点对点的(point-to-point)通道,它仅连接两个进程;或者是一个多路(multiway)通道,它能实现多于两个进程间的通信。在一些情况下,多路通道可以要求每一个进程被赋予一个唯一的地址,它可指定对于任何数据传输的发送/接收进程。

消息传递通信中一个更进一步的差别在于它的两种模式:阻塞模式(blocking)或是非阻塞模式(non-blocking)。如果一个进程必须挂起或阻塞自身在通道上的通信,直到其他进程对它的数据传输做好准备,我们称这样的消息传递通信是阻塞模式。阻塞模式下的通信能有效地促使两个进程在数据传输初始化之前,彼此同步。阻塞模式通信的另一个优点在于它并不要求额外的存储单元来实现通信。然而,因为一个进程必须始终挂起直到其他进程准备好通信,这样阻塞模式下的通信将可能产生性能上的恶化。非阻塞模式的通信并不存在这个问题,因为进程不需要彼此同步传输数据。然而,在这种情况下,额外的存储单元必须隐式地和通道相结合,通常来说,是以队列(queue)方式作为存储单元。发送进程通过通道将要传输的数据写入队列中,接着它可以继续正常的执行,而不管接收进程是否准备好从队列中接收数据。因为两个进程相互独立执行,仅通过队列交换数据,因此在非阻塞模式下的通信通常会导致一个较好的性能。然而,速度上的优势却是以实现队列所需的额外存储的代价来实现的。此外,如果队列长度不够,进程仍然可能在某些点上被阻塞传输数据。例如,当发送进程以一个更快的速率向队列发送数据,而接收进程却以相对较慢的速率从队列获取数据时,最终将引起队列被数据填满,而阻塞发送进程。

79

3.2.7 同步

在一个由几个并发进程组成的系统中,进程很少是完全相互独立的。每一个进程都可以生成一些被其他进程使用的数据和事件。在这种情况下,当进程在交换数据时或某种动作必须被不同的进程同时执行时,我们需要同步化(synchronize)这些进程,一个进程被挂起只有等到其他进程执行到某个点时才能继续执行。一般同步的方法可以分成两类,即与控制相关的

80 同步机制和与数据相关的同步机制。

1. 控制相关的同步

在控制相关的同步技术中,利用行为的控制结构完成系统中两个进程的同步。例如,在3.2.3节中介绍的分叉-结合语句结构(fork-join)是控制结构的实例。图3-9a中显示了行为X将分叉成fork三个并发的子进程A、B和C。在图3-9b中,我们可看出对于行为X,这些不同的执行流是如何通过join语句来进行同步的。它确保了被fork语句派生的三个进程在进程R执行之前全部完成。控制相关同步的另一个例子是使用初始化(initialization)技术。类似于许多硬件描述语言(HDL)一样,进程在系统首次初始化时,在初始状态下得到同步;或者在进程的执行阶段时同步化。在图3-9c的状态图中,可以看到重新进入ABC边界的迁移弧所关联的事件 e 是设计为将所有互不相关状态A、B和C都同步为它们的默认子状态。同理,在图3-9d中,事件 e 使B初始化为为其默认子状态B1(因为AB已经退出,然后重新进入),同时从A中的子状态A1迁移到A2。

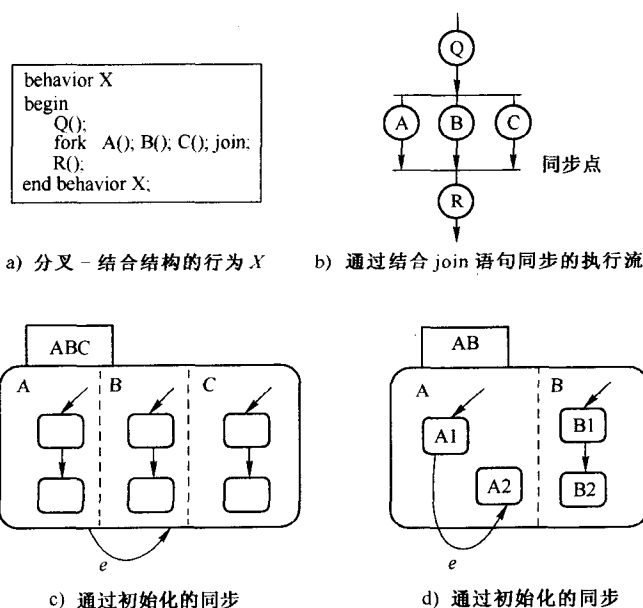


图 3-9 Statecharts 中控制同步

2. 数据相关的同步

除了控制相关同步的这些技术之外,进程也能通过以下两种方法来完成它们之间的通信:共享内存或是消息传递。

(1) 基于共享内存的同步(shared-memory based synchronization)的工作方式是:通过挂起一个进程,直到其他的进程用合适的值更新共享内存。在这样的情况下,共享内存可以表示一个事件、一个数据值或是系统中另一个进程的状态,如图3-10中的Statecharts语言所示。

公共事件的同步(synchronization by common event)要求一个进程必须等待一个特殊事件的发生,而这个事件可以由外部或是另一个进程来产生。在图3-10a中,我们可看到事件 e 是如何分别同步状态A和状态B,并进入其状态中同步子状态A2和B2的。另一种方法是公共

变量的同步(synchronization by common variable),它要求某一进程用一个适当的值更新其变量。图 3-10b 中,当我们给状态 A2 中变量 x 赋值为“1”时,状态 B 被同步到子状态 B2。还有一种方法是状态检测的同步(synchronization by status detection),一个进程在继续执行之前需检查其他进程的状态。在这种情况下,通过事件 e 触发的从 A1 到 A2 的迁移可能会引起 B 从状态 B1 向 B2 的迁移,如图 3-10c 所示。这种同步的方法在我们必须有效表示复杂情况时是非常有用的。这种情况可以是当一个全局异步收发器(universal asynchronous receiver-transmitter, UART)进入接收模式时,需要让传输者的输出端三态化(tristating)。

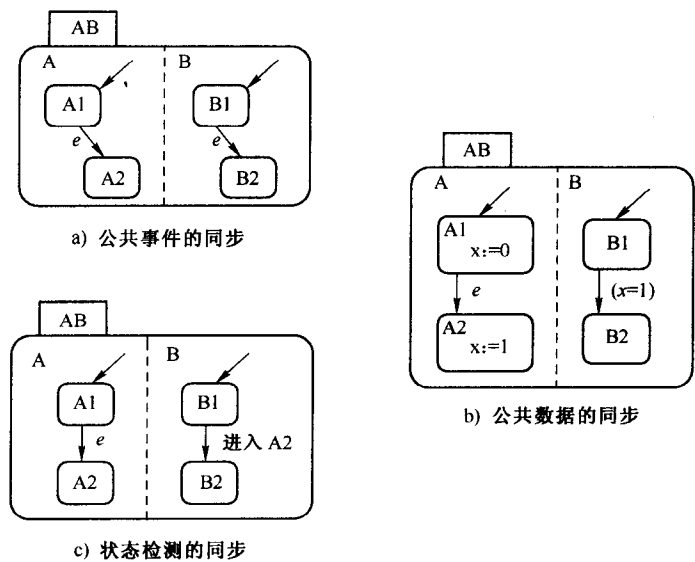


图 3-10 Statecharts 中数据依赖的同步

(2) 通过消息传递同步(synchronization by message-passing)是使用阻塞通信模式的另一种方法。在这种情况下,进程已经准备好首次发送或接收数据,但它必须等待直到另一个进程也准备好通信。例如,CSP 模型能支持这种同步机制。

82

3.2.8 异常处理

通常,某种事件的发生要求一个行为或模式能立刻中断,这样就能阻止其行为进一步更新它的值。因为与行为相关的计算可能很复杂,也许需要花费无限的时间来处理。这样,有必要用称为异常(exception)的一种事件的发生来立即中断其当前的行为,而不必等待计算的完成。当这样的异常发生时,应当显式指定一个应将哪一个控制权交给下一个行为。计算机系统复位和中断等就是这种异常的例子。

通常,传统的有限状态机都有这种异常处理机制。在这样的状态机中,与每一个状态相关动作都假定其执行时间为零,这意味者一个异常的发生将仅仅引起控制交给下一个合适的状态。另一方面,Statecharts 和 PSM 两种模型都允许将控制交给更高层次,而结束低层的计算,以这种方式来支持异常的处理。

3.2.9 非确定性

对于系统中一个特殊的状态迁移或执行的计算行为,偶尔会存在多种实现方式。在这种情况下,系统设计者可能并不希望在描述阶段就限制系统的任何一个特别选择。在概念模型中,这就是一个非确定性的值,它允许设计者对于任何系统执行的动作或计算,指定其多种选择方式。在对行为描述模拟时,可以任意选择其中之一来执行。

在概念模型中,有两种类型的非确定性行为。第一种是选择非确定性(selection non-determinism),即几种选择方案之一的非确定性选择。例如,考虑下面的描述,其中 a 和 b 是两个不同的动作:

```
if (x) then
  do EITHER a OR b
end if
```

83 上面语句的语义是当 x 的值为真时,允许执行 a 或 b 任一种操作,但不允许同时执行两种操作。这样非确定性的例子在 CSP 概念模型中也存在,通过使用保护(guarded)语句指明其非确定性。

第二种类型的非确定性是顺序非确定性(ordering non-determinism),它指的是几个执行的动作间非确定的顺序。例如,考虑下面的语句:

```
if (x) then
  do BOTH a AND b
end if
```

当 x 值为真时,上面的语句调用将同时执行两个动作 a 和 b ,但是它们两者执行的顺序却是以一种非确定性的方式进行的。通过使用这种非确定性,系统设计者可避免在系统描述阶段必须指定其动作精确执行顺序的问题。

3.2.10 时序

在系统描述中,时序是一个反映真实世界实现的重要部分。换句话说,通过指定时序,系统设计者可从系统模拟结果中得到更为真实的信息。一般来说,指定系统的时序信息分为两类:功能时序和时延约束。

1. 功能时序

影响系统描述的模拟输出结果的时序信息全都属于这一类功能时序,因此能把它增加到系统的功能性中。例如在状态图模型中,可决定在模式或状态下所花费的最大时间,以指定其时间间隔。另一个例子是利用 VHDL[IEE88]中的 wait 语句,以及在未来的某个时间里用 VHDL 信号赋值语句更新其信号值,表示时间的间隔。例如,考虑下面的 VHDL 语句:

84

```
a <= 30;
wait for 100 ns;
s <= a + 1 after 30 ns;
```

对信号 s 的赋值语句将在对 a 赋值后的 100 ns 执行。然而,由于在 s 的赋值语句中有 after 语句,信号值将会在对 a 赋值后的 130 ns 时得到更新。

2. 时延约束

对一个系统指定的时延约束可被综合和验证工具利用。例如,在综合过程中一个严谨的性能约束将影响为执行计算的系统资源是如何分配。我们也可通过验证工具检查这些约束是否被满足。这样的约束可能不会影响系统的功能性,从某种意义上来说,它们也不会影响描述的模拟结果。

对于系统而言,有几种时延约束的方式。例如,对于执行时间的约束能够作为行为或行为的一部分给出。如图 3-11a 所示,行为 B 被指定了一个 10 ms 的时延约束。在其他情况下,一个行为可能有相关数据传输率上的约束,它指定在此行为消耗或生成数据时必须达到的一个速率。例如在图 3-11b 中,行为 Q 限制最大速率为 10 Mb/s,生成在通道 C 上要传输的数据,该数据随后将被行为 P 使用。最后,交互事件的时延约束常常被用来指定两个不同事件之间发生的最小/最大时间间隔。例如在图 3-11c 中,当信号 in 下降 50 ns 后,信号 out 上升。

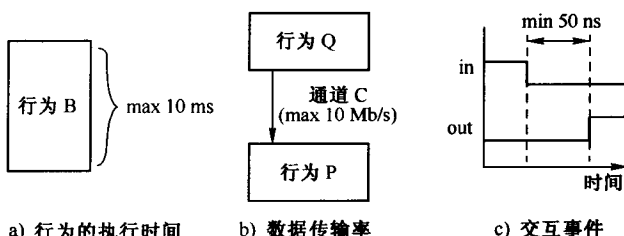


图 3-11 系统描述中的时延约束

在时延约束的描述中,把约束和概念模型中的特殊对象相关联是非常重要的。例如,如果设计者希望在一个行为中限制两个语句间的执行时间时,需要定义一个语句标号,以指定哪两个语句间的时间约束。而在两个 I/O 事件之间,指定约束的最简单方法是使用时序图(timing diagrams)。

3.3 嵌入式系统的描述要求

在前面的章节中,我们已经介绍了不同概念模型中一系列的特性。然而,应当重点注意的是,不同类型的系统可能只需要这些特性中某一类特殊的子集。如果是这种情况,对于一类特殊系统的概念化描述,要求一个具备合适特性子集的概念模型。在这一节中,我们将讨论一些适合于概念化描述嵌入式系统特性的特殊集合。

一个嵌入式系统是指它的行为与其所处的环境相互作用,通常在一个模式集合中使其顺序化,而每一个模式可以代表一个状态或者计算。一般来说,这样的系统会响应外部事件,并以其输入和当前状态的函数来计算它们的输出。嵌入式系统的例子如控制器和电信系统。在下面的列表中,我们讨论了嵌入式系统中几个典型的特性,其中有一些虽然已经在 3.2 节中简要介绍过,但我们还是在此对这些特性重新进行讨论:

1) 状态迁移:嵌入式系统从本质上来说是基于状态的,通常会根据外部事件从一种模式迁移到另一种模式。在某些情况下,这种状态之间的迁移是以一种无规律的方式发生,如图 3-12a 中所示的模式 P、Q 和 R 之间所存在的关系。

2) 行为层次化:当我们把系统功能性看成是由层次化的顺序和并发行为的集合组成时,嵌入式系统就更容易被定义。例如,考虑图 3-12b 中嵌入式系统所实现的功能性,这个系统被

描述成四个不同的行为 P 、 Q 、 R 和 S 。当行为 P 执行后,紧接着是两个并发的行为 Q 和 R 执行,最后才执行行为 S 。应当注意的是,这四个行为中的任何一个行为都有可能是实现(任意)复杂的计算,甚至可以被分解成另外的由顺序/并发子行为组成的附加层次。因为一个嵌入式系统可由几种模式组成,而每一种模式都可以分别响应不同的外部事件,可见,就描述的规模和可读性而言,行为的层次化对于系统的复杂度管理非常重要。

3) 并发性:嵌入式系统通常由一些并发行为组成,而这些行为为了能够实现系统整体的功能,彼此之间需要相互协调。对于这样的系统,任务级和语句级的并发都是基本的要求。

4) 异常:某些事件要求系统给出瞬间的响应。这种情况下,嵌入式系统可能经常会被中断。它们通常结束当前的计算,并迁移到另一种计算模式下以响应这种异常情况的发生。图 3-12c 解释这种瞬间中断的概念。随着事件 e 的发生,行为 P 所执行的计算将被中断,而控制也将转移到行为 Q ,同时 Q 的计算被初始化。

5) 程序设计结构:嵌入式系统执行的某些计算可通过更加自然的方式来表示,如数学表达式,或依赖于程序设计结构(如分支和循环迭代)的算法。

6) 行为完成:在嵌入式系统中,许多行为既不是无限循环的,也不能单独通过外部事件来使之结束。然而,这些行为的执行是在行为达到某个完成点后才能继续的。由于这个特性,我们必须能够指定行为的完成。这一点非常重要,因为我们可能希望以这样的行为完成为基础,初始化另外一个行为的执行。

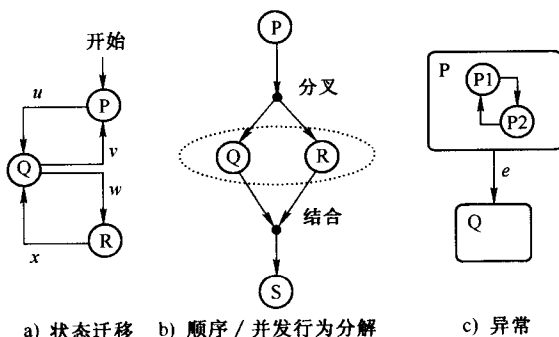


图 3-12 嵌入式系统的特性

3.4 描述语言综述

基于嵌入式系统的这些特性,现在我们将简要地介绍一些被广泛使用的描述语言。我们选择的这些语言包含广泛的描述方式和应用领域,并对当前的设计描述方法具有相当的代表性。在本节中,我们特别关注每一种描述语言是怎样支持嵌入式系统行为描述的。

3.4.1 VHDL

VHDL(VHSIC Hardware Description Language)是由美国国防部提出并发展的,在 1987 年成为 IEEE 的标准。这种语言能帮助设计者进行硬件开发,使设计文档化,并能促进设计的相互交换。随着它的不断发展,VHDL 已经被许多设计单位作为一种描述语言而广泛接受,同时以之为基础开发了一系列的工具,来对一个设计进行捕获(使用图形化前端工具)、模拟、调试、验证和综合。

在 VHDL 中基本的硬件抽象是一个设计实体(design entity),它通常被用来标识和表示一个较大设计中的一部分,执行特定的功能,并指定其输入和输出。使用程序设计语句、数据流、结构或是它们的组合,可描述每一个设计实体的功能性。

在 VHDL 中,结构层次化(structural hierarchy)是通过使用内嵌的块和元件实例化(com-

ponent instantiation)语句来支持的。图 3-13a 中给出了一个模 10 计数器的结构描述。该计数器有两个外部端口 *clk* 和 *cnt*, 在每一个时钟信号 *clk* 上升沿进行计数, 当计数到 9 时重新设置回 0。在结构体声明部分, 声明所有被使用的元件, 指定其形式端口参数 (*Reg_E*、*Add_E* 和 *Cmp_E*)。然后, 元件在结构体部分被实例化 (*Conreg*、*Adder* 和 *Comparator*)。元件之间的互联是由元件端口之间的关联来指定的。例如, 信号 *cnt_out* 被分别映射到元件 *Conreg* 和 *Adder* 的形式端口 *o* 和 *a* 上。这样, 就指明 *Conreg* 的输出端口和 *Adder* 的一个输入端口相连。

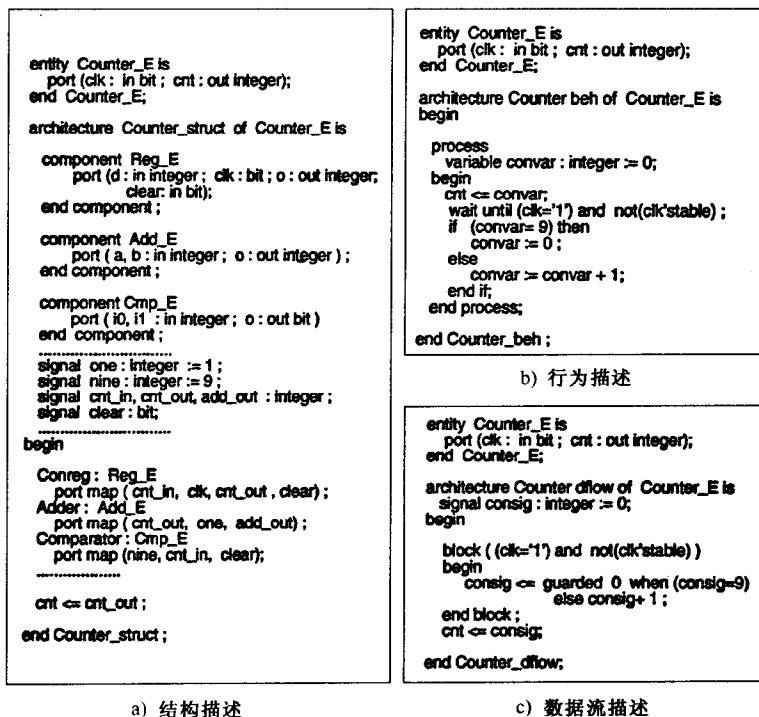


图 3-13 一个模 10 计数器的三种方式的 VHDL 描述

VHDL 支持两级的行为层次 (behavioral hierarchy)。在顶级, 描述被分解成并发执行的进程集合, 以支持任务级的并发。第二级则将这些进程分解为顺序过程。

除了传统的程序设计变量之外, VHDL 还有信号的概念。信号不同于变量, 因为它的值是以时间定义的。这样, 除当前值之外, 信号都有一个输出波形。VHDL 的信号赋值语句支持语句级并发 (statement-level concurrency), 像下面所示的例子:

```

a <= b;
b <= a;
wait on a;

```

这两个赋值同时发生, 并交换信号 *a* 和 *b* 的值。此外, VHDL 中的 *after* 语句允许使用信号赋值语句来使一个信号的值在未来的某个时刻得到更新。这样, 如 3.2.1 节中讨论过的那样, 支

持隐式的语句级并发。

VHDL 也完全支持程序设计结构,因为一个进程是由顺序化语句组成,类似于 ADA 程序设计语言中的特性。图 3-13b 显示了一个简单模 10 计数器的进程描述。VHDL 也提供广泛的适合于高层建模的数据类型,如整型、实型、枚举、物理、数组、记录和指针类型。

90 VHDL 中并发信号赋值的特性允许设计者描述数据流行为,如图 3-13c 中表示的计数器模 10 的数据流描述。

此外,进程间的通信可通过共享内存模型来实现。共享内存模型使用一些可被任何进程赋值的信号线,同时对于其他进程也是可见的。

VHDL 中的同步有两种实现方式。第一种方法依赖于进程的敏感列表(sensitivity list),它确保了当在敏感列表中任何一个信号有事件发生时,进程将被激活执行。例如,考虑一个进程 P ,它被定义为:

```
P: process (start, x)
begin
    .....
end process;
```

根据这个定义,进程 P 将挂起,直到在信号 $start$ 或 x 发生事件后激活。这样使设计者可通过 $start$ 或 x 信号,同步进程 P 与其他进程间的执行。

同步的第二种方法是采用一个 wait 语句,它挂起一个进程,直到它检测到在指定的信号之一上有事件发生,或是满足指定的条件后激活。例如,下面的 wait 语句确保了只有当信号 x 或 y 有事件发生,或者 $start$ 的值为 1 时,进程才会继续执行。

```
wait on x, y until (start = '1');
```

VHDL 语言中的时序描述仅限于功能时序的描述,例如 after 子句(如 $a <= 2$ after 20 ns)指明在未来的某个时间点上信号值得到更新。同理,一个时间子句,例如 wait on start for 100 ns,可以被用来指定在 wait 语句等待的最大时间。VHDL 语句也能通过预定义的表达式 now 评估一个全局时间的当前值。虽然时序特性的第二种类型即时序约束在 VHDL 中并不被直接支持,但是可使用属性来间接指定。

91

与嵌入式系统相关的特性相比,VHDL 语言在一些方面同样也不支持。例如,VHDL 没有一个能响应异常并结束进程的结构。通过使用 VHDL 中的 guarded 并发信号赋值语句,异常仅被部分的支持。在异常处理的方法中,block 语句相关的 guard 表达式将控制 block 中对信号的赋值。

最后,需要注意的是 VHDL 不支持状态迁移。而且,一个真实的行为层次化,其中并发是在任何层次级别中指明的,这种方式也不被 VHDL 支持。

3.4.2 Verilog

Verilog[TM91, SST90]原本是作为一个专用的硬件描述语言(HDL)而发展的,它主要是针对数字系统的描述和模拟这两个方面。然而在 1990 年,Verilog 已经发展到公众领域,因此它也被作为描述语言而广泛使用。

对于设计者来说,Verilog 有很多优点,其中之一是通过允许系统分解为一些层次化的互联模块(module)而支持结构层次化。这些模块都可用两种方式来描述,或者用其他更低层的

模块来说明,或者用一个程序来指定其行为。

在 Verilog 中同样支持行为层次化。从某种意义上来说,在层次化中的任何一个级别,进程都可用分叉-结合语句派生并发的子进程,或是由一些子过程组成。这些进程的描述用类 C 的语法,通过程序设计结构(programming construct)fork-join 来实现。另外,也可以使用持续赋值(continuous assignment)语句捕获数据流(dataflow)行为。

在 Verilog 中,通信是用共享内存模型来实现的。通过使用连接线把模块、寄存器和内存的端口相互连接起来,建立进程之间的通信。Verilog 也能以几种方式来支持同步,因为控制同步可以使用分叉-结合结构来实现,观测一个事件值改变的事件控制语句也常常用来同步不同进程之间的计算。例如,语句:

```
@ (negedge) clock #10 q = d;
```

可确保在时钟信号 *clock* 在变成低电平的 10 个时间单位后,把信号 *d* 的值赋给信号 *q*。另外,我们也能使用 Verilog 的 *wait* 语句来实现同样的效果,如:

```
wait (clock = 0);
#10 q = d;
```

在 Verilog 中通过建立门与网时延的模型支持时序(timing)描述。这意味着对于每一个上升、下降和翻转的时延,Verilog 允许设计者指定其最大、最小和典型的数值。另外,Verilog 同时允许设计者决定在一定时延后,赋值语句中的其信号值将得到更新。例如,在赋值语句 *#10 q = d* 中, *q* 的值将在时钟信号 *clock* 下降沿的 10 个时间单位后得到更新。

Verilog 通过使用 *disable* 语句也能处理一些异常(exception),它能禁止顺序语句中指定的块的执行,把控制流转移到这个块后面的语句中。

最后,还应指出的是 Verilog 不支持状态迁移的描述。

3.4.3 HardwareC

HardwareC[KD88]专门被设计为面向综合的硬件描述语言。它以 C 程序设计语言[KR78]为基础,同时具备适用于硬件描述的额外定义的语义和结构。正因为这些特性,在 Hercules 行为综合系统[DK88]中选择这个语言作为最有效的方式来指定其输入。

在 HardwareC 中,一个系统的描述可以由一个单一层次的并发进程组成,它们彼此之间相互通信。这些进程可以包含在一个层次化的块中。在这种情况下,这些块和它们之间的互联构成了结构层次(structural hierarchy)。

我们也可在 HardwareC 中用进程的方式指定任务级并发(task-level concurrency)。每一个进程通过一系列顺序操作给出一个算法的描述。这些操作可以是以 C 语言程序设计结构中的子集的来描述。每个进程一旦完成便又从头开始执行。语句级并发(statement-level concurrency)通过使用并行复合语句(见 3.2.1 节)指定。

HardwareC 中不同进程间的通信(communication)通过共享内存或消息传递模型的方式来实现。例如,端口传递(port passing)是假设一个共享的媒体已存在,如连接信号线或内存,它们将连接已声明的端口。设计者可以在通信的进程边界或它们包含的块中定义这些端口,并使用显式的 HardwareC 命令来读、写或三态化一个全局端口。用来进行通信的协议被指定为进程描述中的一部分,如图 3-14a 中所示,图中进程 Main 通过端口 *n_p* 提供值 *n* 给进程 Fac-

92

93

94

torial,并且接收端口 r_p 的结果。另一方面,消息传递(message passing)通过声明相互通信的进程/块之间的通信通道后,使用显式的发送-接收结构来实现数据的传输与同步。在这些通道上的通信可被同步化或是被阻塞。使用这些结构,设计者仅需要指明用握手协议的方式实现在通道上数据的传输,而相关的硬件可以利用综合工具来自动综合生成。例如,在图 3-14b 中,可以看到进程 Main 使用通道 $ch1$ 来发送值 n ,并使用通道 $ch2$ 来接收结果。

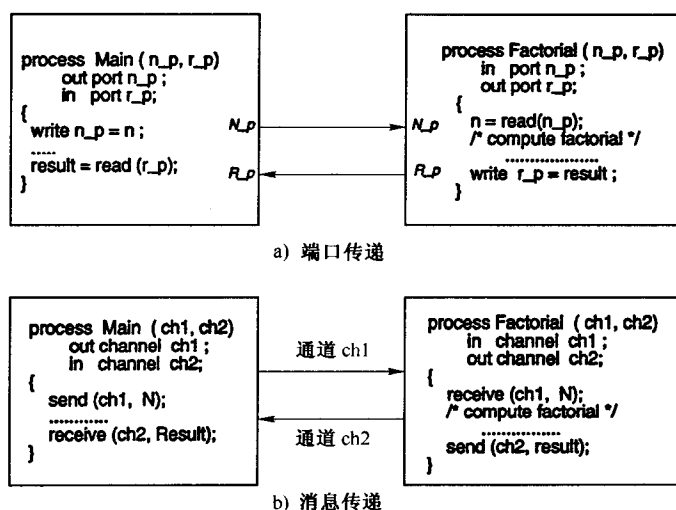


图 3-14 HardwareC 中的进程间通信

为了使 HardwareC 中的两个进程同步(synchronize),设计者可以使用阻塞消息传递方式。通过使用一个 msgwait 结构,可以检测到已经在通道中等待的未决消息。通过使用这个结构,设计者可以指定一个进程一直等待来源于另一个进程的适当信号,这个信号可以用一个二进制位的消息来表示。

HardwareC 中一个有趣的特性是对于不同模型(例如块、进程、过程和函数)具备指定参数化描述的机制,即模板(template)。通过对这些模板的形式参数提供整数值的方式来使之实例化,同时,我们利用这种特性来描述库元件,比如用位宽、输入端口数等形式参数来描述加法器和乘法器。设计者可实例化一个模板,以便被综合工具利用的绑定信息为方式,来指定它的一个特定实例。例如,对于一个加法模板的特定实例,可绑定为一个特定的加法操作,使系统设计者可在描述级上指定其资源的共享情况。

在 HardwareC 中,两个语句之间的时序约束(timing constraint)的指定可通过在每个语句加一标识 tag 或 label 实现,而后使用标识来指定约束。此外,HardwareC 也可指定资源约束,以决定在综合设计时,对于一个给定的模型,能有多少个实例数(类似于在高层次综合过程中的分配任务[GDWL91])。

95

最后,我们需要指出的是 HardwareC 不支持数据流行为、状态迁移以或异常的描述。

3.4.4 CSP

在多处理器机器上运行程序的时候,传统的程序设计语言具有较大的局限性。为了克服这一局限性,C.A.R.Hoare 在 1978 年提出了通信顺序进程语言(Communicating Sequential Process,CSP)[Hoa78]。顾名思义,CSP 允许设计者用一系列并发进程来描述程序,通过结构

来简化对通信和进程间同步的描述。另外, CSP 作为一种程序设计语言, 也能用来描述硬件系统[Aul91]。

一个 CSP 程序含有一系列命令(command)。利用并行命令 parallel, 一个进程可以被派生为在层次任何级别的若干子进程(任务级并发), 这提供了对行为层次化的支持。在 parallel 命令中所有进程都会被并发执行, 当所有这些进程都终止后, parallel 命令才结束。而且, 这里提到的进程本身就是一系列命令, 并可以含有属于进程自身的 parallel 命令。

在 CSP 中, 可以用程序设计结构来描述进程, 而子程序通过协同例程来实现, 这意味着子程序也是通过进程来实现, 它和调用进程并发执行。递归子程序可通过一个进程序列来模拟, 序列中的每个元素都表示一个级别的递归。

CSP 中的控制结构通过命令 guarded 来实现, 该命令含有一个监视或条件表。当监视表中的所有条件都满足的时候, 这个命令列表将被执行。为了指定 guarded 命令中仅一个命令被执行, 可以使用 alternative 命令来进行描述。这样, 在 C 语言中的“if”语句

```
if ( a > b ) max = a;
else      max = b;
```

可以被 CSP 中的 alternative 命令描述如下:

```
[ a > b → max := a [ a ≤ b → max := b ]
```

96

在使用 alternative 命令时, 可能会出现多个监视条件同时满足的情况。在这种情况下, 需要仲裁的 guard, 来决定选择和执行哪些命令, 产生非确定性行为。

在 CSP 中没有全局变量。因此, 并发进程间的通信只能通过指定显式 input 和 output 命令, 使用消息传递的机制来实现。两个进程间的通信只有满足下列标准的时候才会发生:

- 1) 在第 1 个进程的 output 命令中, 指定数据传输的目的地是第 2 个进程。
- 2) 在第 2 个进程的 input 命令中, 指定数据接收的来源是第 1 个进程。
- 3) 在 input 命令中, 接收数据的目标类型和 output 命令的表达相符。

两个进程间通过 input/output 命令的通信是一种阻塞式通信, 它构成了 CSP 中唯一的同步机制。

值得注意的是, CSP 也有自身的局限性, 它没有提供对结构、状态转换、时序、数据流行为和异常处理的支持。

3.4.5 Statecharts

状态图表 Statecharts[Har87, DH89]语言最初是为描述反应系统(reactive system)而设计的, 这种系统通常是由事件驱动, 受控制支配的系统, 用于航空控制系统和通信网络等。状态图表在传统有限状态机的基础上进行了扩充, 增加了层次、并发和通信三个新的元素, 能够很好地对目标进行描述。

为了介绍状态图表语言, 图 3-15 中描述了一个通用异步接收机发送机 UART。状态图表中的基本对象是状态, 两个状态之间的转换是由事件和条件的组合所决定。

状态图表支持行为化层次, 它允许将任意描述分解为状态的层次化。具体说来可以有下面两种实现方式:

- 1) OR(顺序)分解: 将一个状态分解为包含顺序的子状态和转换弧的状态机。例如图

97

3-15 中, 状态 *tx_mode* 含有 *idle* 和 *transmit* 两个顺序的子状态。

2) AND(并发)分解: 将一个状态分解为互不相关的子状态, 这样只要父状态被激活, 所有子状态都能同时被激活。在图 3-15 中, 不相关状态(并发状态)用虚线分开, 也就是说顶层状态 *top_level_uart* 含有三个并发状态: *transmitter*、*receiver* 和 *uart_mode*。在状态图表中, AND 分解指明了任务级并发性。

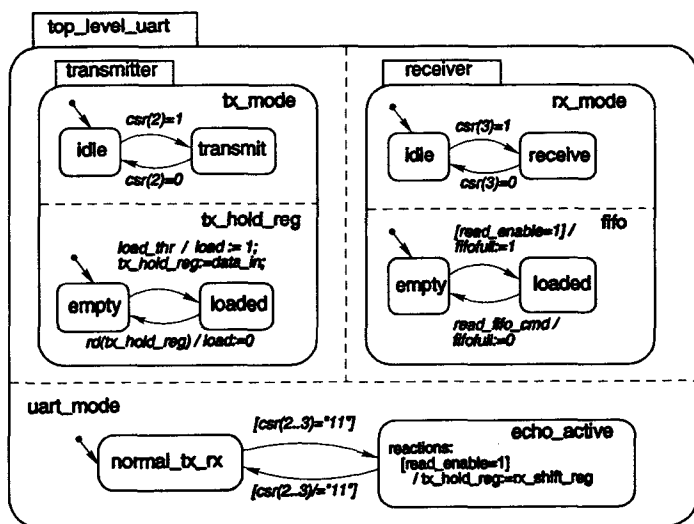


图 3-15 Statecharts: UART 的部分描述

在传统有限状态机中, 状态一般都是单一层次并顺序执行的, 可能出现状态数呈指数增长的情况。状态图表允许将状态分解为顺序和并行子状态, 以避免这种情况的发生。每个动作都被认为是没有时延的计算过程, 能够和状态转换弧和状态相关联。动作和状态之间的联系有两种方式: 一种是当系统处于该状态的时候不断执行动作, 另一种是当进入或者离开该状态的时候执行动作。状态图表允许在层次中不同级别间进行状态迁移。

在状态图表中, 功能时序是通过使用超时转换弧来指定的, 这些弧定义了系统在这个状态下能够花费的最多和最少的的时间。

状态图表通过广播机制来实现通信。当系统中的任何部分产生新的事件、变量更新或者发生了状态转换, 这些信息都会被立刻传到状态图表的其他部分。该语言的一个有趣的特性是能支持多种同步机制, 包括初始化、公共事件、公共数据和状态检测等方式, 这些在 3.2.7 节中有详细介绍。

为了确定一组子状态中的起始子状态, 状态图表还支持系统历史的概念。例如在历史记录(enter-by-history)里, 对一组子状态, 控制将首先进入最近访问过的子状态。

此外, 状态图表还能描述非确定性行为。状态图表中的状态可以同时发出两条弧, 选择哪条弧, 即转换到哪一个状态具有非确定性。

状态图表的缺点在于它不支持程序设计结构, 也没有提供对结构、行为完成和数据流行为的支持。

3.4.6 Argos^①

Argos [Mar91, Hal93] 是一种用于描述反应系统的图形化同步语言。Argos 的基本进程是有限自动机,它能接收和发送信号。Argos 采用和状态图表相同的图形约定,但是在语言上有几点不同之处。首先 Argos 将状态间的约束限制于同一层次,因此能够将一个系统分解为具有更好的模块化的若干子系统。另外,状态图表中将所有事件都向整个系统传播,为了减小这类事件的规模,Argos 中引入了局部事件(local event)概念,将事件的处理局限于一个特定的子系统或模块中。最后,Argos 采用了自终止机制(self termination),可以指定行为完成。

Argos 的缺点在于它不支持程序设计结构,也没有提供对结构或数据流行为的支持。

3.4.7 SDL

规范与描述语言(Specification and Description Language, SDL)[BHS91]由 CCITT 标准化,主要用于电信领域中的协议描述。

SDL 描述主要含有层次化的数据流图,其叶结点是状态机。SDL 能够描述层次化结构。在 SDL 中,描述的目标对象是系统,它能够用层次化的块(block)来定义,块是 SDL 中主要的结构概念。每个系统含有一个或多个块,这些块都通过通道(channel)与其他块以及系统边界相连接,如图 3-16 所示。这些通道实质上构成了信号传递的通路。对这些块可以进一步进行分解,最后得到一个层次化的树形描述,其根结点为整个系统。每个叶结点块可以含有一个或多个进程。

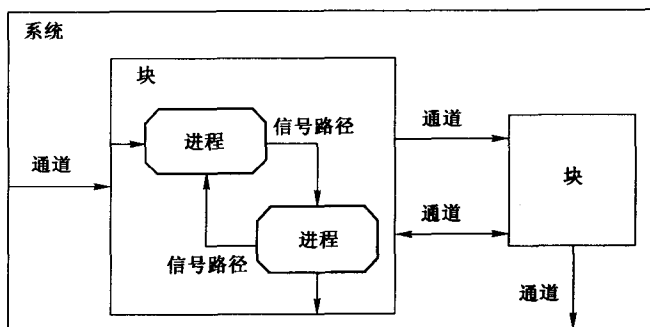


图 3-16 通过层次化分解将 SDL 描述分解为系统、块和进程

SDL 对行为层次化的支持是有限的,这是因为在它的叶结点块描述中含有单一层次的进程。另一方面,SDL 支持进程的状态机顺序分解。这些进程可以在系统初始化的时候被激活,也可以在系统生存期里被其他进程激活。一旦进程建立后,只有达到停止结构才会终止,这意味着 SDL 支持行为完成。

SDL 的每个进程都含有一个状态机,能够支持状态转换。变量可以在进程中声明和赋值,进程自身可以存在于状态内或在状态间执行转换。在转换之间,进程可对变量进行处理、做出决定、建立新的进程实例、向其他进程发送信号,以及激活或者复位产生时序信号的时钟。

进程间以及进程和块的边界间具有信号通路(signal route),通过在这些通路上传递信号,

① 本小节原书未标章节号,译者添加章节号,并将后面的章节号顺延。——译者注

可以实现通信进程。此外,SDL 中每个进程都具有一个输入队列,用来对输入信号进行缓冲。队列中的信号在进程处理后将队列中移出。SDL 中的进程可以检查并等待信号满足指定的值,这样可以实现同步。在进程中声明时钟对象,在预设时钟间隔到达之前来产生时钟信号,这样可以在 SDL 中进行时序描述。

最后,SDL 不支持数据流描述、程序设计结构或异常处理。

3.4.8 Silage

Silage 语言[HR92, Hil85]的提出是为了解决数字信号处理(digital signal processing, DSP)系统中的相关问题。DSP 系统接受连续的输入数据值流,并对其进行一系列操作。Silage 是适合支持这种系统的应用语言,它仅指定处理数据值所需的功能应用,而不涉及变量及变量的赋值。

Silage 的主要优势在于对数据流进行描述,如数据驱动的并发操作。Silage 的表达式表示数值流,例如对于表达式 $a + b$, a 和 b 分别表示数值流,而不是通常程序设计语言里的变量或者数组元素。Silage 程序接收的一组输入值以同步方式到达。Silage 表达式的结果同样是数值流的形式。一个 Silage 程序由一系列定义组成,将新的值定义为已有值的函数。因为这些定义并不代表对变量的赋值,各语句间是相互独立的,因而与顺序无关。

在 Silage 的循环中,每个流的元素都依赖于这个流的前一个值,这在一定程度上实现了功能时序。时延符号@用来表示流的前一个值,如下面的语句:

```
d = d@1 + 1;
```

在这个流中的值用 d 来表示,每次的值都比前一个值高 1。

Silage 具有数组结构(array constructor),能够精确地表达向量元素。另外,求和与求最大值等归约运算可对整个数组进行操作。在监视条件的基础上,条件表达式能够在一系列表达式中选取一个表达式。Silage 还具有部分去除(decimate)和内插(interpolate)等对于流的操作,可分别对不同信号的采样率进行增加或减少。Silage 中的函数表达了一组定义,通过宏展开的模式来实现。Silage 中不允许递归或非静态边界的循环。

由于 Silage 是作为一种应用语言提出的,它不支持一般程序设计语言结构。此外,如上文所述,它没有变量、赋值操作和状态转换,也不支持层次化行为和异常处理。

3.4.9 Esterel

Esterel[Ber91, Hal93]是一种用于开发反应系统的同步程序设计语言。迄今为止,它已经用于描述控制器、通信协议、人机接口和系统驱动等领域。

Esterel 语言是基于完美同步假设(perfect synchrony hypothesis),即程序中状态的控制、通信和计算都不需要时间。程序设计结构在 Esterel 中得到了很好的支持,层次化结构也能通过层次化模块指定,这些模块都具有定义好的接口,并能表示其他模块。Esterel 中提供了程序顺序化分解以及通过并行语句来实现并行分解的特性,这些特性构成了对层次化行为的支持,在语义上和分叉-结合语句类似。

Esterel 中的通信是通过信号的广播来实现的,也就是说一个进程发出的信号将被其他所有进程所检测到。Esterel 提供了多种处理异常的方法,例如 watching 语句:

```
do
  <语句>
watching s
```

do-upto 语句:

```
do
    <语句>
upto s
```

以及 trap 语句:

```
trap t in
    <语句>; exit T
|| await s; exit T
end
```

当含有信号 s 的事件发生的时候,这些语句都能够终止它们所包含的语句块的执行,在这一功能上这些语句是相同的。

同步在 Esterel 中也有多种表达方式。例如 await 语句,如“await s .”,保证当前进程等待指定信号的发生。另外,如果我们指定了在 watching 或 do-upto 语句中的语句块只含有停止(halt)语句,那么它们也可以用来同步。

然而,Esterel 不含有描述状态转换、时序和数据流行为的结构,因此并完全不适合于描述嵌入式系统。

3.5 SpecCharts

在 3.3 节中对嵌入式系统描述的特点进行了概括,然而前面提到的所有语言都无法满足这些特点。本节将介绍 SpecCharts 语言,它专门针对这些特性的处理,能够直接获得概念上的模型。

3.5.1 语言描述

SpecCharts [NVG91a, NVG92, GVN93] 的基础是第 2 章中介绍的程序状态机(program-state machine, PSM),同时也是 VHDL 的扩展。如图 3-17 中所示,SpecCharts 中的基本对象是行为,它直接和 PSM 模型中的程序状态相对应。

SpecCharts 对层次化行为的支持体现在它通过层次化的行为来描述系统,其中的每个行为或是复合行为或是叶行为。

复合行为(composite behaviors)可以被层次化分解为一系列并发的子行为或一组顺序行为。对于前者,当该行为激活的时候,所有子行为也都被激活;而对于后者,一次只能激活一个子行为。在图 3-17 中,行为 B 和 X 都是复合行为,其中 B 含有并发子行为 X 、 Y 和 Z , X 含有顺序子行为 $X1$ 和 $X2$ 。对于顺序分解的复合行为,子行为表中的第一个就是初始行为,在父行为激活时将控制信号传递给初始行为。在图 3-17 的 SpecCharts 描述中,当行为 X 被初始化时,其子行为 $X1$ 将激活。另一方面,对于并发的子行为,它们的顺序则没有任何关系。

在图 3-17 中, $X1$ 、 $X2$ 、 Y 和 Z 都是叶子行为。叶子行为位于层次化行为的底部,其功能性都使用 VHDL 顺序语句,通过程序设计结构来指明。

图 3-17 中的描述 $X1$ 和 $X2$ 位于其父行为 X 中,SpecCharts 通过这样的方式将子行为描

```

entity E is
  port ( P : in integer; Q : out integer );
end E

architecture A of E is
begin
  behavior B type concurrent subbehaviors is
    type int_array is array (natural range <=>) of integer;
    signal M : int_array(15 downto 0);
  begin

    X: (TOC, true, complete);
    Y: (TOC, e3, complete);
    Z: ;

    behavior X type sequential subbehaviors is
    begin
      X1: (T1, e1, X2);
      X2: (TOC, e2, complete);

      behavior X1 type code is .....
      behavior X2 type code is .....
    end X;

    behavior Y type code is
      variable max : integer;
    begin
      max := 0;
      for J in 0 to 15 loop
        if (M(J) > max) then
          max := M(J);
        end if;
      end loop;
    end Y;

    behavior Z type code is .....

  end B;
end A;

```

图 3-17 一个简单的 SpecCharts 描述

述嵌套在父行为中。类似地,子行为描述 X 、 Y 和 Z 都包含于行为 B 之内。

SpecCharts 也支持状态迁移。从某种意义上来说,我们可通过一系列转换弧来表示子行为间的先后顺序。在 SpecCharts 中,用三元组 $\langle T, C, NB \rangle$ 来表示这样的弧,其中 T 表示迁移类型, C 表示触发该迁移的事件或条件, NB 表示迁移的下一个行为。如果迁移不需要条件,那么就使用默认的“真”值。

和 PSM 模型一样,SpecCharts 有两种类型的迁移弧。第一种是完成迁移弧 (transition-on-completion arc, TOC),当迁移弧的源行为计算结束,并且满足相应条件后,则转向该弧指向的行为。对于一个叶行为,当其所含最后一个 VHDL 语句执行完后,整个行为就结束,此时所有变量和信号的值都将更新为最后值。对于顺序分解的行为,结束的标志是生成一条指向预定义结束点的迁移弧,在该弧表示中的下一行为域以 *complete* 作为参数名。如图 3-17 中,行为 X 结束的标志是其子行为 $X2$ 完成操作,满足事件 $e2$,并将控制流由 $X2$ 转向 *complete* 完成点 (该转换弧为 $X2: \langle TOC, e2, complete \rangle$)。最后,对于并发分解的行为,需要其所有 (或选择的一定子集)子行为都结束后,父行为才完成。如图 3-17 所示,例如,当行为 B 的并发子行为 X 和 Y 都结束,控制转向完成点 (相应的转换弧为 $X: \langle TOC, true, complete \rangle$ 和 $Y: \langle TOC, e3, complete \rangle$) 后,行为 B 才完成。需要注意的是,在这个例子中,行为 B 是否结束和其子行为 Z 的执行状态无关。然而,子行为 Z 会受到行为 B 的影响,也就是说当 X 和 Y 都转换到完成点,也就是 B 完成的时候, Z 也会终止。

另一种迁移弧是即时迁移弧(transition-immediately arc, TI),这种弧在满足相关弧条件的时候立即进行状态转换,而不管源行为是否结束其计算。如图 3-17 中的弧 X1: $\langle TI, e1, X2 \rangle$,当事件 $e1$ 满足的时候,行为 X1 立刻终止,控制转向行为 X2。换句话说,即时迁移弧能有效地终止源行为所有低级的子行为。超时弧(timeout arc)是一种特殊的 TI 弧,它在到达指定时间间隔后进行转换,由行为需要被激活的时间所确定。

在 SpecCharts 中,每个时刻只能有程序状态的一个子集向外传送计算。整个程序状态中只有根结点是一直处于激活状态,它代表着整个系统。值得注意的是,SpecCharts 的执行方式和 VHDL 类似,其中的活动行为和 VHDL 中的基本一样,除了行为中不含隐式循环外。换句话说,每个行为都将一直执行,直到碰到 wait 语句才挂起,两个连续 wait 语句将没有时延,所有信号都将在 δ 时间内更新。不活动的行为和 VHDL 中进程的挂起一样,将被忽略,其所有的信号源都将关闭。

在所有顺序子行为都完成后,如果任何 TOC 迁移弧都不满足条件,系统则会进行隐式的等待状态,直到满足某个条件。SpecCharts 采用确定的描述,从某种意义上来说,离开一个子行为的迁移将放在描述列表的优先位置。通常, TI 弧比 TOC 弧有较高的优先级,并且它在整个层次化中具有最高的优先级。

SpecCharts 中的行为可以含有 VHDL 声明,如类型、信号、变量和过程,其作用域包括了该行为的所有子行为。

SpecCharts 有两种支持同步的方法。其一是用 wait 语句来检查事件和条件,这和 VHDL 中的方法一样。如下面的语句

```
wait until (start = '0') and (not start' stable);
```

将挂起当前行为,并等待信号 *start* 的下降沿。第二种方法是使用一个 TI 迁移弧,从行为 *B* 指回自己,以此来将其所有并发子行为同步到初始状态。 [107]

和 VHDL 类似,SpecCharts 也支持层次化结构,能够将系统或者系统的一部分封装为实体。每个实体都可以声明其端口,通过信号和其他实体的端口相连。如图 3-17 所示,整个层次化行为封装为实体 *E*,声明的端口指定该实体的接口。

在 SpecCharts 中,通信是通过变量和信号来实现的。顺序行为可以对同一个变量进行读写,而并发行为间的通信需要使用信号。在 SpecCharts 中,还可以定义适当的发送-接收过程来指定消息传递通信。由于 VHDL 中允许对子程序名重载,用户可以对通道中不同类型的数据都定义发送-接收过程。

SpecCharts 中可以在信号赋值中使用 wait 语句和 after 子句,实现时序描述,这和 VHDL 的处理方法一样。此外,还可以用超时 TI 弧来对时序进行描述,它指定一个行为的执行时间上限。

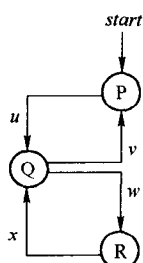
3.5.2 用 SpecCharts 描述嵌入式系统

上一节已经对 SpecCharts 做了一个简单的介绍,接下来将针对 3.3 节中概括的嵌入式系统描述要求,说明如何用 SpecCharts 来描述嵌入式系统。换句话说,SpecCharts 能够从自身的语言结构上直接映射为一个嵌入式系统的概念模型。

图 3-18、图 3-19 和图 3-20 分别说明了如何用 SpecCharts 来描述状态迁移、层次化行为和异常。通过 TOC 和 TI 弧将行为进行序列化,以此来处理状态迁移。一个行为可以含有顺序或

并发子行为,这样可直接支持层次化行为。通过 TI 弧可以直接捕获异常。需注意的是,图中左边的概念模型和右边的 SpecCharts 描述是直接相对应的。

108



a) 期望功能

```

behavior MAIN type sequential subbehaviors is
begin
  P : (TOC, u, Q);
  Q : (TOC, v, P); (TOC, w, R);
  R : (TOC, x, Q);

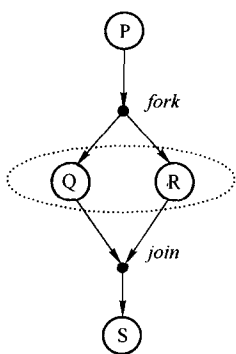
  behavior P ----
  behavior Q ----
  behavior R ----

end MAIN;

```

b) SpecCharts 描述

图 3-18 描述状态迁移



a) 期望功能

```

behavior MAIN type sequential subbehaviors is
begin
  P : (TOC, true, Q_R);
  Q_R : (TOC, true, S);
  S : ;

  behavior P ----

  behavior Q_R type concurrent subbehaviors is
  begin
    Q : (TOC, true, complete);
    R : (TOC, true, complete);

    behavior Q .....
    behavior R .....
  end Q_R;

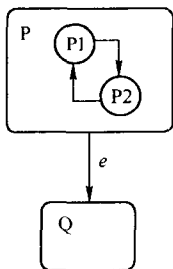
  behavior S....
end MAIN;

```

b) SpecCharts 描述

109

图 3-19 捕获顺序/并发表行为分解



a) 期望功能

```

behavior MAIN type sequential subbehaviors is
begin
  P : (TI, e, Q);
  Q : ;

  behavior P ....
  behavior P1
  .....
  behavior P2
  .....

  behavior Q
  .....
end MAIN;

```

b) SpecCharts 描述

图 3-20 捕获异常

此外, SpecCharts 是在 VHDL 语言的基础上建立的, 它能直接使用 VHDL 的顺序语句来支持程序设计结构, 这一点是显而易见的。最后, 行为完成可以通过 TOC 弧和完成点来处理。

3.5.3 等价图形化表示

在图 3-17 中,我们引入了 SpecCharts 语言,并把它作为文本描述方式。然而,对状态转换,使用图形化的状态迁移图更容易理解。为此,在这个观点的基础上,SpecCharts 语言提供了等价图形化的表示,以一种可视化的形式来捕获状态迁移结构。如图 3-21,我们给出了和图 3-17 中文本 SpecCharts 描述等价的图形化表示。在某些情况下,SpecCharts 的图形约定和 [Har87] 中的相类似。

在图 3-21 中,矩形框代表实体,圆角矩形框代表实体中的行为,用虚线分隔并发的子行为,用箭头表示迁移。对顺序分解的行为,用实心倒三角指定第一个子行为,如 X1 是行为 X 的初始子行为。顺序分解行为的完成通过指向完成点的迁移弧来指定,其中完成点用实心正方形来表示,并位于该行为内部。如行为 X 中,子行为 X2 在满足条件 e2 的时候转到完成点。

TOC 和 TI 的迁移弧有不同的图形表示法。TOC 弧从源子行为内部的实心正方形出发(如标明 e2 和 e3 的弧),而 TI 弧从源子行为的边界出发(如标明 e1 的弧)。

110

最后,系统是用文本描述还是图形描述并不重要,相反,这种结构能更好地体现语言的优势。例如,一个状态机可通过一个状态表的文本方式来描述,也可以使用状态迁移图进行描述。文本和图形描述都有各自的优点,但最重要的是状态迁移自身的概念,它在某种环境下是很有价值的。至于使用文本还是图形方式来对系统进行描述,已经超出了本章的范围。

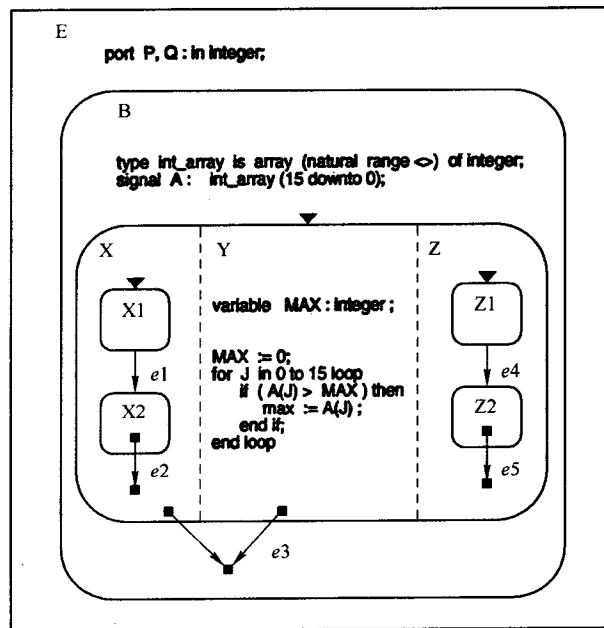


图 3-21 等价的 SpecCharts 图形表示

3.5.4 语言的可扩展性

SpecCharts 语言可以从多个方面进行扩展。首先,设计者经常需要将一组顺序描述的行为分解为一组并发子行为。对于这种情况,将分叉连接语句的能力应用到叶行为上很有用。其次,系统设计者可能希望对一个能连续执行并发任务的行为进行描述,如定义一个信号 enabled,

111

当信号 *power* 为“on”的时候(即, *power* = 1)并且外部信号 *reset* 引脚为低(即, *reset* = 0)时, *enabled* 变为 1。根据 *power* 和 *reset* 信号,在系统生存期内定义 *enable*,这样的能力是非常有用的。如下面的 VHDL 并行信号赋值语句:

```
block
begin
    enable <= power and (not reset);
end block;
```

112

另外一种有用的扩展是支持参数化行为。对于一个多数行为都类似的大型、规则系统,只需要定义少数几个参数,并指定适当的参数值,就能方便地对所需的行为进行实例化。而现在的情况是,即使系统中有 *N* 个完全相同的行为,它们中的每一个都需要用 SpecCharts 语言进行描述。

3.6 结论和发展方向

本章的目的在于展示概念模型特性和用来描述这些特性的语言结构之间的一种直接映射的关系。由于缺乏一一对应的关系,对于系统描述的任务是非常困难的,容易发生错误或是不完全的描述。在这一章中,我们首先给出了通用概念模型中的多种特性,接着介绍了八种语言,说明这些语言是如何支持这些特性的,如图 3-22 所示。对于嵌入式系统特定实例,我们给出了 SpecCharts 语言中的结构能够容易地描述 PSM 模型的特性,并能够很好地适应嵌入式系统的描述。语言和模型匹配得越好,描述所需的时间就越少,描述中的错误也越少,另外,描述本身的综合性和适应性也就得到了提高。

语言	嵌入式系统特性					
	状态迁移	行为层次性	并发性	程序结构	异常	行为完成
VHDL	○	○	●	●	○	●
Venlog	○	●	●	●	●	●
HardwareC	○	○	●	●	○	●
CSP	○	●	●	●	○	●
Statecharts	●	●	●	○	●	○
SDL	●	○	●	○	○	●
Silage	-	-	●	-	-	-
Esterel	○	●	●	●	●	●
SpecCharts	●	●	●	●	●	●

● 完全支持

○ 部分支持

○ 不支持

- 不可应用

图 3-22 各种语言对嵌入式系统概念模型特性的支持

然而,应当意识到,随着语言描述所处层次的提高,语言表达能力也随之提高,由此带来的负面影响使设计实现变得更加困难。例如,即使广泛地使用类似于 VHDL 语言,综合工具也只能处理语言结构的有限子集。如果语言不仅仅是作为文档,未来发展的重点将是它的可综合性问题。特别应当指出的是,如果某种语言特性没有硬件的支持,就无法应用于系统建模,除非设计者不需要对其直接进行综合。此外,系统复杂度在不断提高,使得通过模拟验证 (verification by simulation) 的方法是不可行的,因为描述中有大量的信号和事件,因而在模拟中可能的情况数目也都变得十分庞大,这无疑使模拟验证变得不可管理。因此,我们需要研究开发一种能自身就支持形式化验证技术的语言,这样才有可能对系统的特性进行自动验证。

113

3.7 练习

1. 3.2 节中提出了概念模型的特性,C 程序设计语言支持其中的哪些特性?
2. 列举下面系统所需的概念模型特性:
 - (a) 数字信号处理器系统;
 - (b) 通信协议;
 - (c) 数据库系统;
 - (d) 指令集处理器。
3. 从同步、通信和时序几个方面对比 VHDL、HardwareC、Statecharts、CSP、SDL 和 SpecCharts 等语言,列出每种语言相关的结构(提示,参考[NG93])。
4. 根据图 3-5 中的层次化有限状态机(FSM),生成平面有限状态机。二者相比,层次化有限状态机具有什么优点?
5. 状态图表 Statecharts 概念模型可处理状态迁移和层次化的特性。考虑一个具有 H 层的状态图表,在每一层都采用 OR 分解为 S 个顺序子状态,每个子状态都发出一条简单迁移弧(即每一层有 S 个迁移弧)。假设没有层内迁移,求状态总数和迁移总数。
6. 如果将上题中的状态图表展平为一个有限状态机,又有多少个状态和迁移?
7. 下面的 VHDL 顺序语句计算 a 、 b 和 c 中间的最大值:

114

```
process(a, b, c)
begin
    if (a > b)
        then max <= a;
        else max <= b;
    end if;
    wait on max;
    if (c > max)
        then max <= c;
    end if;
end process;
```

用 VHDL 并行信号赋值重写上面的数据流语句。

8. 对比 VHDL 的信号和变量赋值语句的语义。在下面的代码中,如果 a 和 b 的值分别是 10 和 20,在语句执行后的终值分别是多少?

(a) 信号赋值:

```
a <= b;  
b <= a;  
wait for 1 ns;
```

(b) 变量赋值:

```
a: = b;  
b: = a;
```

9. 分别用 Verilog、HardwareC、CSP、Esterel 和 SpecCharts 等语言对图 3-13 中的模 10 计数器进行描述。

115

10. 考虑下面的语句:

```
a = 1;  
x = 3;  
y = x + b;  
b = a + 1;
```

并假定 b 和 x 的值都为 2。如果上面的描述是一组顺序语句, 执行后 b 和 y 的值分别是多少? 如果是数据流语句, b 和 y 的值又是多少?

116

第 4 章 系统描述举例

在前面的章节中,我们知道能够描述系统特性的最好语言应当能够最容易抓住系统自然模型的特性。本章我们将给出一个小而完整的嵌入式系统的实际系统描述,以便显示出概念模型和系统描述语言之间的匹配所带来的优势。

4.1 引言

本章中,我们将用一个电话应答机的例子来演示系统概念化和可执行功能描述的开发。我们从一个对嵌入式系统功能性的自然语言描述开始,逐步进入到 SpecCharts 语言的 PSM 模型的开发。我们将看到用一个可执行语言描述系统比起用英语等传统自然语言描述的优势。

我们也将通过几个实验来说明:利用 SpecCharts 描述嵌入式系统比用 VHDL 描述在获取时间、理解时间和功能错误方面都有所减少。减少的原因主要由于 SpecCharts 语言对 PSM 模型的支持。我们还给出一个能显示出 SpecCharts、VHDL 和 Statecharts 描述之间差别的实验。最后,我们将通过一个实验来证明利用可执行描述进行自顶向下的设计并不会降低系统的设计质量。

117

4.2 电话应答机

这里作为举例用的系统是当前市场上正在使用的一种电话应答机的控制器。控制器的运行环境如图 4-1 所示,我们将描述这个环境和控制器的接口。

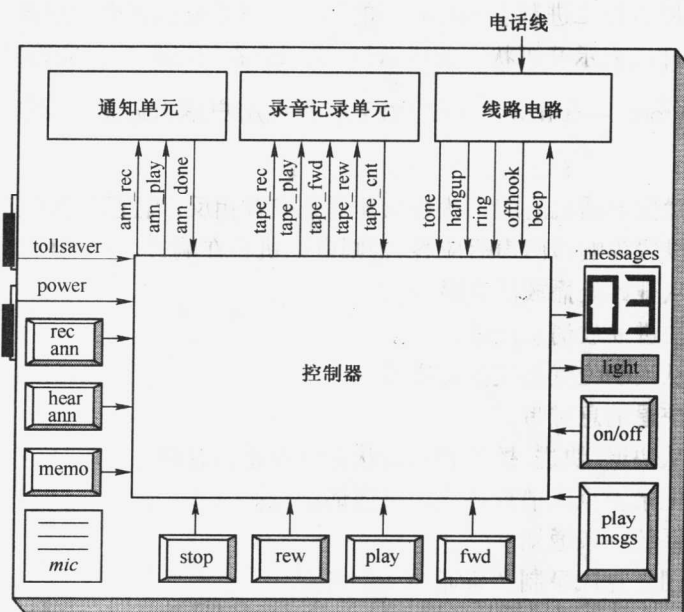


图 4-1 应答机控制器的环境

118 通知单元(Announcement unit)用来记录和传播一个较短的外发通知,接口包括三个端口:

ann_rec: 指挥该单元从麦克风记录电话机主人的通知。
 ann_play: 指挥该单元播放通知。
 ann_done: 指示控制器,通知已经结束。

录音记录单元(Tape unit)用于在磁带上记录消息并重复播放,接口包括五个端口:

tape_rec: 指挥该单元把来电记录到磁带中。
 tape_play: 播放录音。
 tape_fwd: 快进录音。
 tape_rew: 快退录音。
 tape_count: 将当前的录音位置作为整数传给控制器,0代表录音的开始。

标准的线路电路(Line circuitry)用于呼叫应答,来电检测,挂断检测,按键音调解码和产生蜂鸣。控制器与该电路的接口有五个端口:

tone: 代表从线路上检测到的用4位二进制编码的按键音调。
 hangup: 检测线路上的挂断。
 ring: 检测线路上的来电。
 offhook: 指示线路电路进行呼叫应答。
 beep: 指示线路电路在线路上产生蜂鸣。

显示模块(display)用于显示当前的消息数量和应答机的开/关状态,接口由两个端口组成:

message: 用5位二进制表示接收到的消息数量并显示在电话机屏幕上。
 light: 开启表示开机状态的指示灯,表示应答机已经建立应答准备。

119

九个触摸按键(touch-sensitive button)用于应答机用户编辑通知和接听消息,对应的端口如下:

on_off: 置位电话机的应答状态从开到关或者相反,当置于开状态时,除非将电话机设置为响铃两声后应答,否则电话机将在响铃四声之后应答。
 play_msgs: 从开始处播放消息磁带。
 fwd: 快进播放消息磁带。
 play: 从当前位置播放消息磁带。
 rew: 快退消息磁带。
 stop: 从快退、快进、播放和录音状态下停止消息磁带。
 memo: 记录麦克风消息替代记录通话记录。
 hear_ann: 播放一条通知。
 rec_ann: 用麦克风录制一条通知。

上述端口的功能和通知单元、录音记录单元上端口的不同之处在于它们仅表示用户的请求。只有当控制器适当地配置了通知单元或录音记录单元的端口之后,这些动作才能真正发生。

该应答机还有两个开关：

- power: 电话机的开关。当开关处于“off”位置时,电话机对电话呼叫和所有的按键动作均忽略,并且关闭消息计数显示和电话机状态指示灯。该端口不同于 on/off 端口,这个端口只有当 power 开关处于“on”位置时才指示电话机是否对呼叫做出应答。
- toll saver: 响铃状态指示。当应答状态为“on”时,电话机在记录至少一条信息之后应当在响铃两声后应答(相反的情况是四声后应答)。

120

4.3 用 SpecCharts 进行系统描述

我们现在将用 SpecCharts 来获取应答机控制器功能性的一个可执行描述。前面各章所介绍的图形结构将在下面的图示中用来表示组合行为。控制器的功能性描述在附录 A 中用自然语言描述;下面介绍其中一部分。

首先我们指定系统的接口。图 4-2 给出了接口描述的一部分,该描述可以比较容易地从 4.2 节的接口自然语言描述中得到。

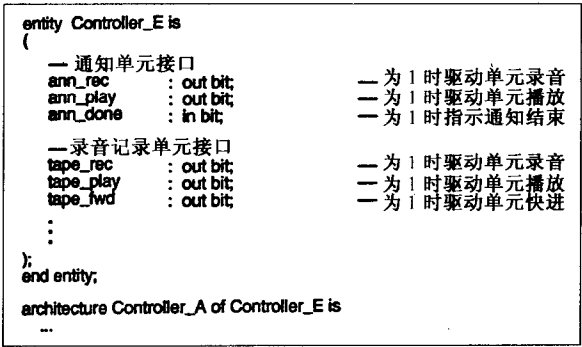


图 4-2 应答机控制器的接口

接下来我们把控制器的行为分解成若干简单行为,进而这些行为又可以继续分解为更加简单的若干子行为,直到这些子行为可以用顺序语句描述。

控制器行为:自然语言描述称“当 power 开关处于‘off’位置时,电话机忽略线路呼叫和所有的按键”。因此,在抽象的最高层,控制器始终处于两种模式之一:“on”或者“off”。我们将控制器分解为两个顺序的子行为,系统开(SystemOn)和系统关(SystemOff),如图 4-3 所示。如果在系统关的状态下 power 变成 1,则控制器进入系统开状态。一条 TI 弧表示这种状态迁移。类似地,也有一条弧表示当 power 变成 0 时,控制器回到系统关的状态。这样的 on/off 分解在许多系统中都很常见。

系统关的行为:当系统为“off”时,它处于完全空闲,因此系统关的状态不包含任何功能,我们将其指定为仅包含单条 VHDL 描述“null”的叶子行为。

系统开的行为:系统处于“on”状态的功能性描述可以分解为两个部分:“响应外部按键和开关”以及“响应线路呼叫”。我们据此将系统开分解为两个子行为:按键响应 Respond-

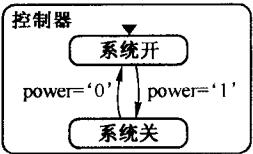


图 4-3 控制器的最高层次视图

121

ToMachineButton 和线路响应 RespondToLine,如图 4-4 所示。为了确定它们的转换关系,我们注意到描述状态:“电话机的按键的响应比起线路响应的优先级要高;因此按下任何外部按键的同时也将中止任何当前活动。”这段描述表明电话机的按键将触发即时响应。因此在图中用一条从线路响应到按键响应的 TI 弧来表示任何按键按下所导致的状态迁移。为了保证可达性,我们创建了一个新信号, any _ button _ pushed, 表示所有电话机按键信号的逻辑或。

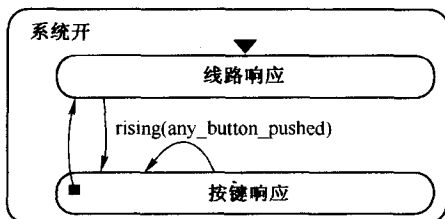


图 4-4 系统开的行为

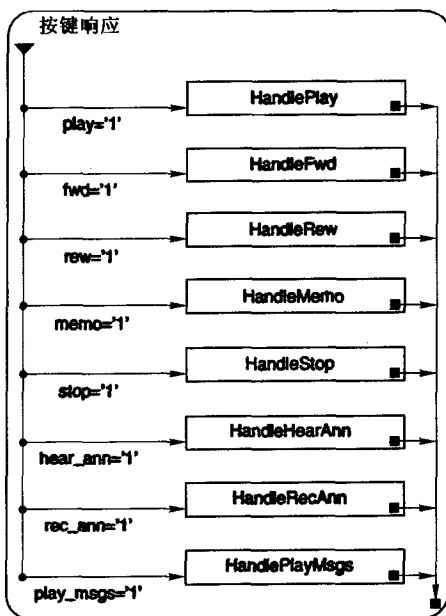
有一种可能的情况是当控制器仍在响应第一次按键的时候另一个按键又被按下。例如,如果“快退”按键被按下,系统的响应是将磁带倒回到开始处。然而,用户可以在倒带到开始处之前按下“停止”。因此我们在按键响应状态上增加一个 TI 弧自环来表示对最新按键的响应。为了避免在单独按下一个按键的处理过程中,系统发生退出或者重入按键响应状态的非期望行为,我们指定状态转换只能发生在 any _ button _ pushed 信号的上升沿处。

在自然语言的描述中没有明确说明回到线路响应状态的条件是什么。由于没有例外能够导致这种状态迁移,我们假设该迁移仅发生在对按键的响应结束之后。因此添加一条从按键响应到线路响应的 TOC 弧,默认条件是“true”,表示当按键响应状态结束之后,立即回到线路响应状态。

按键响应的行为:不同的按键要求有不同的响应处理(具体的响应内容参见附录),我们通过一个过程来描述每一个响应,并且利用 if-then-else 语句来调用每个特定按键所对应的处理过程,如图 4-5a 所示。注意,当一段代码执行完毕,按键响应的行为结束,系统将如前所述返回线路响应状态。

```
behavior RespondToMachineButton
type code is
begin
  if (play='1') then
    HandlePlay;
  elsif (fwd='1') then
    HandleFwd;
  elsif (rew='1') then
    HandleRew;
  elsif (memo='1') then
    HandleMemo;
  elsif (stop='1') then
    HandleStop;
  elsif (hear_ann='1') then
    HandleHearAnn;
  elsif (rec_ann='1') then
    HandleRecAnn;
  elsif (play_msgs='1') then
    HandlePlayMsgs;
  end if;
end;
```

a) 顺序语句



b) 等价子行为分解

图 4-5 按键响应的行为

我们也可以与前面相同的方法将按键响应分解为若干顺序子行为。每个子行为对应一个特殊的响应,而弧线指出哪一个子行为将被激活,如图 4-5b 所示。TOC 弧从每一个子行为出发指向按键响应行为的同一个特定的完成点。两种描述在功能上等价,因此系统建模者可以自由选取最方便和可读性最强的描述方法。

线路响应的行为:自然语言描述中提到,当电话机响应线路呼叫时将执行以下三个任务之一:“监测线路的响铃”,直到响铃计数的值到达规定的数目;“正常应答活动”,播放通知同时记录呼叫信息;“远程操作应答活动”,用户(假设为机主)可以通过播一系列的按键来收听到呼叫留言。我们认为将线路响应分解为两个主要的子过程监测器(Monitor)和应答器(Answer)是最容易理解的,如图 4-6 所示。正常应答和远程操作应答的区别将在后面的应答器行为中描述。

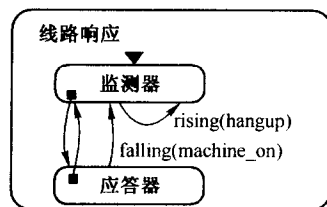


图 4-6 线路响应的行为

当正确的响铃计数被监测到之后,监测器结束,同时应答器激活,因此从监测器到应答器之间添加一条 TOC 弧。自然语言描述中提到在呼叫信息被记录之后,“电话机挂断并重新监测响铃”。因此我们再从应答器到监测器添加一条 TOC 弧。我们还发现:“如果在电话机开始应答之后‘on/off’按键被按下,则当前的活动将被中止,电话机重新监测电话线路。由于有人可能在收听留言时拿起电话并按下‘on/off’来关闭电话机,而转入与其他人通话,因而该功能可以用于屏蔽呼叫。”用一条从应答器指向监测器的 TI 弧来表示,转移条件为 machine_on 的下降沿。如果当前电话机状态为“on”,machine_on 信号在描述中为“true”。

自然语言描述中没有明确提到这种情况:如果电话开始响铃但是呼叫者已经在被应答前挂断,监测器将重新启动。我们在监测器上加入自环 TI 弧来捕获这种行为,如果检测到挂断则执行该状态迁移,令监测器重启。我们可以等价地将该挂断行为指定为监测器的一部分,加一条 TI 弧的方案可以导致监测器行为更为简单。

监测器行为:监测器监测线路上的响铃计数直至达到要求的数量。自然语言描述在多个条件的基础上给出了三种不同的可能性:“当电话机处于‘on’状态,电话机一般在四声响铃之后应答。然而,如果‘toll saver’端口处于‘on’并且至少记录了一条信息,电话机将在响铃两声后应答。Toll saver 允许机主通过电话来判定是否有任何消息被记录。如果电话响铃三声,说明没有任何消息,机主可以挂断电话从而避免了长途通话的费用。有时候机主可能在离家之前忘记了打开电话机,因此即使处于‘off’状态,电话机仍将在响铃十五声以后应答。”

虽然自然语言描述没有明确说明,这些条件也将在电话响铃之后改变。举个例子,假设要求应答的响铃数为十五,且当前呼叫已经响铃五次。假设此时 machine_on 信号变为“1”,使得要求应答的响铃数变为四;该呼叫也应当立即被应答。

监测器可以被描述为两个并行的子行为:一个用于计算应答前等待的响铃数,另一个用于响铃计数并且在监测到要求的响铃数量后应答。我们将监测行为分解为等待维护响铃 MaintainRingsToWait 和响铃计数 CountRings 两个子行为,如图 4-7 所示。声明一个信号 rings_to_wait,由等待维护响铃子行为写入,而由响铃计数子行为读入。当达到要求的响铃计数后,监测行为结束。因此从响铃计数到监测状态的结束点增加一条 TOC 弧。

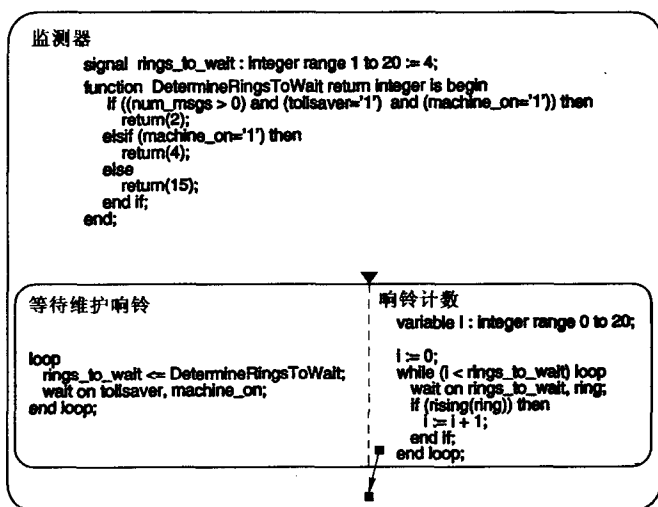


图 4-7 监测器行为

监测器也可以通过顺序语句描述为一个单独的算法,系统建模者可以自由选择最方便和可读性最强的方式来描述它的功能。

应答器行为:应答活动的功能描述如下:“一旦电话机应答线路呼叫,就播放通知。通知播放完毕将产生蜂鸣,同时线路上的信息被记录下来直到挂断或者超过最长的信息记录时间,之后电话机挂断并继续监测响铃呼叫。如果在播放通知的过程中监测到用户的挂断,电话机立即挂断并且不记录消息。如果按键音 1 被监测到,则不管当前正在播放通知还是正在录制信息,电话机都将立即进入远程操作模式。”

我们将应答器分解为四个子行为:播放通知 PlayAnnouncement、记录消息 RecordMsg、挂断 Hangup 和远程操作 RemoteOperation,如图 4-8a 所示。只要没有异常发生,系统将顺序执行前三项操作,转换的完成使用 TOC 弧。一种异常的情况是在播放通知状态下发生挂断操作。在这种情况下,用一条 TI 弧表示立即迁移。在记录消息的过程中发生挂断认为是一种正常情况而不是异常。另一种异常的情况是在播放通知和记录消息的过程中出现“0001”的按键顺序,这种情况出现后系统将从播放通知和记录消息的状态通过 TI 弧立即迁移到远程操作状态。挂断状态结束后状态将迁移进入应答完成点。

播放通知行为:通知的播放包括三个简单步骤,可以表示为图 4-8b 所示的三个顺序语句描述。

记录消息行为:记录一条信息非常简单,用顺序语句描述的步骤如图 4-8c 所示。在一秒钟的蜂鸣结束之后,消息被记录直到发生挂断或者 100 秒超时。第二声蜂鸣在消息结束时产生,同时消息计数增加。

注意,描述中说明了在一秒钟的蜂鸣过程中出现呼叫者挂断的可能性。如果呼叫者确实进行了挂断,则 hangup 信号为‘0’,于是行为结束且不执行记录和增加消息数的语句。

远程操作行为:关于远程操作的描述如下:“远程操作的第一步是检查用户的身份号码 ID,接下来四次按键序列的值将和内部存储的值进行比较。如果二者不匹配,电话机将挂断电话;否则,电话机进入基本命令模式,该模式下可以执行几种基本命令。”

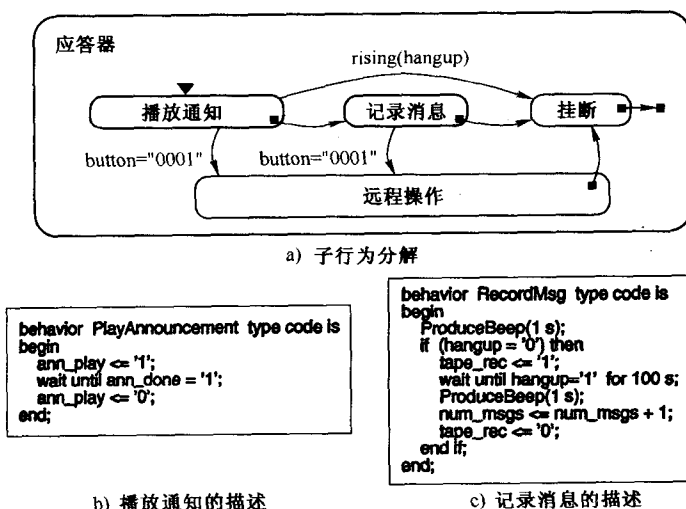


图 4-8 应答器行为

这里将远程操作分解为两个顺序子行为：号码检查 CheckCode 和命令响应 RespondToCmds,如图 4-9a 所示。在检查了由用户输入的表征身份的四位数字码后,当且仅当号码匹配进入命令响应状态。我们引入一个布尔值信号 code_ok,号码检查状态在判断输入号码正确后将该信号赋值为“true”。这里将引入两条 TOC 弧,一条从号码检查指向用户命令响应,状态转换的条件是 code_ok 的值为“true”;另一条从号码检查指向远程操作的完成点,转换发生的条件为 code_ok 为“false”。如果在号码检查过程中发生挂断,则通过一条 TI 弧转换到远程操作的完成点。

128

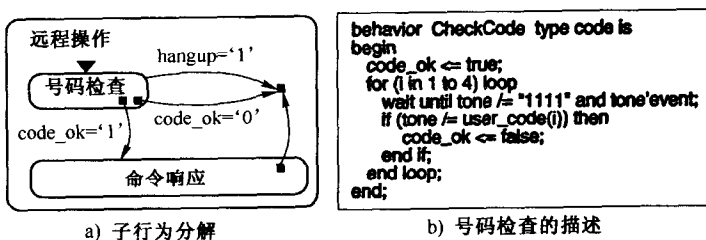


图 4-9 远程操作行为

号码检查行为:代码检查的行为可以用图 4-9b 来描述。从程序可以看出,如果连续四次拨号输入值和存储在 user_code 中的数值匹配,则 code_ok 为“true”,否则为“false”。注意,即使检测到不匹配的值,算法仍将继续执行直到四次拨号音的结束。该过程防止监测到错误按键时的立即挂断,电话机通知一个非法用户其拨号音不正确。

命令响应的描述方法和上面类似,为简便起见此处不再赘述。图 4-10 概述了分解后控制器的子行为和状态变迁。声明和叶结点的描述在这里省略;关于应答机控制器的完整 SpecCharts 描述请参见附录 B。

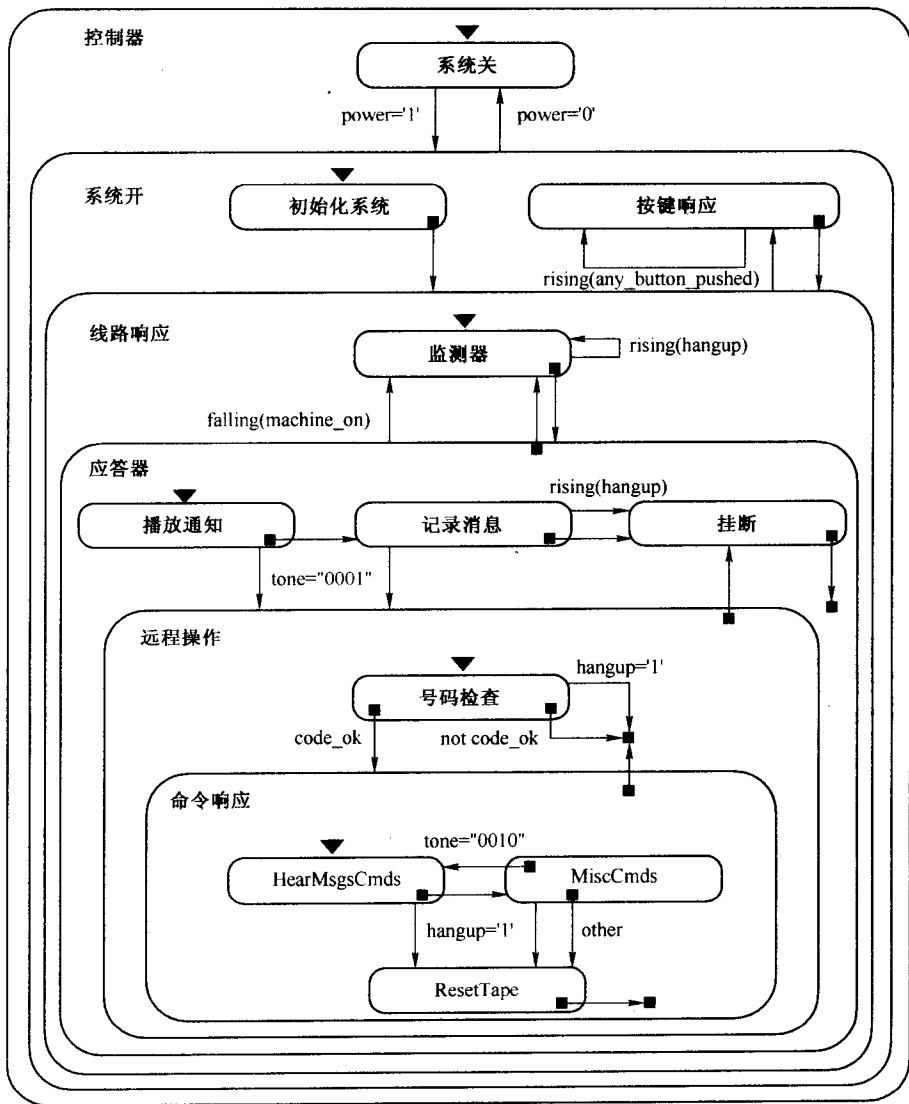


图 4-10 应答机控制器的系统描述

4.4 测试用例举例

为了用模拟的方法验证控制器的功能,我们生成一个测试用例。一个测试用例将待测元件实例化为一个测试模块,得到模拟元件在实际工作状态下的输入值,并且保证在输出端得到正确的预期值。如图 4-11 所示,首先置位 `power` 为 '1',接着按下 on/off 键来打开电话机,如果操作成功将在短时间内设置 `on_off` 为 '1'。我们生成四次响铃,同时保证 `offhook` 为 '1',判断呼叫是否被应答。如果呼叫没有得到应答,模拟过程会打印一条错误信息。这类测试用例可能包含几千行代码。

```

entity testbench_E is
...
end entity;

architecture testbench_A is
begin

    component Controller_E
    port
        ( tone : in bit_vector(0 to 3);
          hangup : in bit;
          ring: in bit;
          ...
        );
    end component;

    -- 初始化组件, 为端口加入全局信号
    ...

    process

        procedure PressButton(but: out bit) is
        begin
            but <= '1';
            wait for button_hold_time;
            but <= '0';
        end;

        procedure Ring is
        begin
            ring <= '1';
            wait for ring_hold_time;
            ring <= '0';
            wait for ring_hold_time;
        end;

    begin
        -- 打开应答机
        power <= '1';
        tollsaver <= '0';
        PressButton(on_off);

        -- 加入一个呼叫
        Ring; Ring; Ring; Ring;
        wait for 1 s;
        assert (offhook='1') report "ERROR1: did not answer";
        ...
    end process;
end architecture;

```

图 4-11 应答机测试用例的举例

131

4.5 可执行系统描述的优点

在检测 SpecCharts 和其基本模型 PSM 的描述能力之前,先看一下在上面的例子中,相对于自然语言而言,用可执行语言来描述系统的优点。可执行语言的主要优势是使设计结果减少功能性错误。少数功能性错误的出现的原因主要在于:自然语言的描述虽然易读,但是这种易读性却带来降低精确性的代价。可读性较好的自然语言描述通常由一连串的情况描述组成,而对同样功能的一个可执行描述却是由包含处理上述情况的算法组成。例如,下面是一段选自前面小节内容的自然语言描述:

“on/off”按键用于将电话机从“开”状态置为“关”状态,或者相反。当电话机处于“on”状态,一般在四声响铃之后应答。然而,如果“tollsaver”端口处于“on”并且至少记录了一条信息,电话机将在响铃两声后应答。

上述最后两句话的每一句都描述了对特定情况的一种反应。假设我们的初衷是明确描述所有可能情况下的行为。我们必须加入以下这些描述:

在铃声计数的过程中,machine_on 或者 tollsaver 可能改变,此时必须重新计算电话机等待所需要的响铃数,并且继续记录响铃数直到达到该数值。

如果我们试图用这种方式明确地列举出所有可能的情况,则系统的自然语言描述将会变

得庞大而难读,或许看上去更像法律条文而不像一个容易理解的文档。为了更加精确地覆盖所有可能出现的情况,我们应当将一段描述当作算法来看,比如:

132 首先确定需要等待的响铃数,接下来重复以下过程直到检测到必需的响铃数:

我们等待响铃或者 on_off 或 tollsaver 信号的变化。如果监测到响铃,则增加当前的响铃计数。如果 on_off 或 tollsaver 信号变化,则重新判定电话机等待需要的响铃计数。

这样的描述写起来并不自然且难于理解。

不同于自然语言,可执行语言对上例这样的算法描述则更胜一筹。因此我们看到自然语言描述和可执行描述一般用于不同目的且互为补充。自然语言的描述偏重强调将一般的行为描述为一系列情况,留下太多需要读者推测的部分,然而可执行描述对行为的详尽描述不会留下任何解释上的变化空间。一个经过充分证明的可执行描述可服务于上述两种用途。

通过创建可执行系统描述,设计者必须在实现一个系统之前就考虑到细节的功能性。除此之外,在获取系统描述之后,设计者可在脑海中构建各种可能情况并且判断功能是否正确。例如:我们可以假设如下问题:“当五次铃响之后且 machine_on 为 0 时会发生什么?”监测模块的代码可以精确地回答该问题:呼叫得到应答。然而自然语言描述就不能直接回答这样的问题。

采用可执行描述的另一个好处是可以用模拟的方法来进行进一步验证系统的功能。所有这些前期的努力都将极大降低在设计后续阶段出现的功能性错误的数量,这同时也意味着耗时的设计修改次数将减少,同时大大降低整个系统的设计周期。

4.6 PSM 模型的优势

133 应答机控制器的例子说明,从 SpecCharts 语言及其 PSM 模型,如 3.3 节中描述的嵌入式系统的六个特性是如何很容易就得到的。

4.6.1 层次性

系统采用自上而下的分解方式。例如,在最高层系统被认为具有两个行为模型,系统关和系统开。系统开又被分解为几个子行为。由于人们习惯一级一级地思考从而控制复杂度,自然语言描述也趋向于层次性。SpecCharts 本身对层次性的支持使得可以将自然语言的概念层次描述方便地映射为 SpecCharts 的结构。

4.6.2 状态迁移

多数系统行为都是顺序的,从一个行为转为另一个,然而状态迁移却是非结构化的。例如,应答行为包含四个子行为和六个彼此直接可能的迁移。这种非结构化的迁移很容易映射为 SpecCharts 中的状态迁移结构,因为从任何行为都可以直接迁移到其他行为。这种迁移用单独的顺序语言很难描述,就像大多数程序设计语言和硬件描述语言所要求的那样。用顺序语言来描述非结构化的迁移需要用到 goto 语句,但是为了保证程序的可读性和模块化,大多数语言不支持该语句。

4.6.3 程序设计结构

描述某种行为的最简单方式是通过算法。例如,号码检查行为可以用一个带条件分支和循环的七行算法来描述。例子中的算法由一系列操作写成,和大多数算法类似,因而可以很容

易地映射为 SpecCharts 中的顺序程序设计结构。

134

4.6.4 并行性

我们看到监测器行为被分解为两个并行的子行为,维护等待响铃(Maintain Rings Towait)和响铃计数(Count Rings)。如果不支持并行性,则并行行为将不得不顺序描述,这种情况下,行为必须足够小到适合顺序描述,顺序化较大的行为通常会导致非常复杂且可读性很差的描述。由于任何行为在 SpecCharts 的任何层次上都可被分解为顺序或并发结构,例子中的并行性可以很容易地映射为 SpecCharts 中一个并发分解行为。

4.6.5 异常处理

在各个层次级别上都有很多异常情况需要处理。一次挂断操作需要作为特殊情况立即处理,同时涉及远程操作、应答和线路响应。线路上的拨号音需要应答的立即处理。machine_on 信号的改变需要线路响应的立即响应。电话机上的一次按键也需要系统开状态的立即响应。最后, power 信号的改变也需要控制器的立即响应。

使用 TI 弧结构可以容易地处理异常。任何时候都会发生很多可能的异常情况,因而要在这些异常中指定优先级是很困难的。然而,层次性可以简化对优先级的指定。如果同时发生很多事件,则 SpecCharts 中高度最大的 TI 弧具有优先响应权。

4.6.6 完成

由于系统采用层次式的分解,在很多情况下,我们在描述一个子行为之前,在子行为之间建立基于子行为完成的迁移。例如,在线路响应状态,我们在监测器完成时建立从监测器到应答器的迁移。在后面将会看到,监测器行为可以被描述为一系列顺序语句或者一组并行的子行为。定义监测器的完成来简化层次分解的复杂性,而不管监测器被指定为顺序描述还是一组子行为。

135

4.6.7 状态分解和编码的等价性

如前所述,一个行为可以被描述为子行为或者程序代码。从层次化的角度来看,这两种方法在功能上是等价的。系统建模者可以自由选用可能的最好方式。在例子中,我们既可将按键响应和监测器描述为程序,也可以描述为顺序或者并发的子行为。利用这些丰富而灵活的选择,编写一个好的系统描述,除了科学性也同样要有技巧性,类似于软件工程。

4.7 实验

从上面的例子可以得到利用 SpecCharts 来得到嵌入式系统描述优势的直观感受。没有它,得到系统描述的时间将增加,对系统功能性的理解将会减少,同时功能性错误的发生可能更多。我们将用几个实验来定量地进行说明。实验将对采用 SpecCharts 和 VHDL 语言得到嵌入式系统的描述进行比较。

4.7.1 系统描述的获取比较

这个实验的目的是说明利用 SpecCharts 描述需求时,对获取时间和描述错误的减少。将建模者分两组,对象是同一个实例系统的自然语言描述。两个小组分别使用 VHDL 和 SpecCharts 进行描述。下面对两个小组的描述过程所花费的时间和描述错误的数量进行比较。

实例系统是一个空中交通报警和防碰撞系统[LHHR92]。之所以选择该系统,一方面它

136

代表一个现成的嵌入式系统,另一方面它的文档可以从外界渠道获得,于是降低了实验不公平的可能性。由于时间限制,只从系统功能中抽取了一部分进行系统描述。三名建模者使用 VHDL,另外三名使用 SpecCharts。

图 4-12 所示是两组建模者系统描述获取的结果比较。VHDL 小组的建模时间是另一组的平均 2.5 倍。此外,两个 VHDL 的规格描述中都存在一个重大的控制错误,该错误导致系统对外部事件的响应严重变慢。这个错误给 VHDL 小组的建模者指出后,他们又试图修正他们的系统描述,结果在指定的时间内只有一名建模者修正了该错误。SpecCharts 由于支持状态迁移和异常处理,被证明更加有效。

	VHDL	SpecCharts
平均描述时间(分)	40	16
建模者数量	3	3
第一次不正确描述的数量	2	0
第二次不正确描述的数量	1	0

图 4-12 系统描述获取的统计

4.7.2 系统描述的理解比较

这个实验的目的在于说明 SpecCharts 系统描述在易理解性方面相比 VHDL 系统描述的优势。给予一个小组的建模者一个系统的 VHDL 描述,给予另一个小组 SpecCharts 描述;每个小组都将回答几个关于系统功能的问题。接下来将比较回答正确的个数和每个小组对系统功能理解的时间。

选取的样例是一个以太网协处理器(ethernet coprocessor)[GD92]的一部分,该例的 HDL 系统描述可以从外界渠道获得。我们手工创建了一个与之功能等价的 SpecCharts 描述,并将 VHDL 和 SpecCharts 描述分别给予三名建模者。每个人理解系统描述的时间都记录下来。在理解之后,每个人都回答十四个关于系统的问题,比如“当使能信号跳变为低位后将发生什么?”“任何输入数据将发送多少个前导字节?”“变量 v 的用途是什么?”

137

VHDL 系统描述小组的建模者对一般行为的理解时间是平均时间的三倍。此外,他们平均有两个错误回答,然而使用 SpecCharts 的小组正确回答了所有的问题。

4.7.3 系统描述的量化比较

为了量化 SpecCharts、VHDL 和 Statecharts 规格描述之间的差异,本实验将一个系统分别用三种语言描述,接着分别比较了规格描述的几个不同方面。

样例选择的是本章描述的电话应答机。分别用 SpecCharts、VHDL 和 Statecharts 从自然语言描述中获取系统描述。创建了两个 VHDL 的版本,其中一个版本利用嵌套模块和通过控制信号的进程通信来维护层次性,将在 5.6 节描述;另一个版本将层次结构展开为平面的程序状态机,接着描述为一个单独的进程。

图 4-13 所示是实验的结果,平面 VHDL 描述的程序状态数较少,只存在叶子程序状态,然而换来的代价是 4 倍左右的获取时间和连接弧数量的增加。原因如下:在层次化的模型中,一条位于高层的弧可以明确表示:一条连接弧条件为真要求从一个状态迁移到另一状态,而并不关心系统当前所处的叶子状态。在平面模型中,这样的一条弧必须把每一个叶子状态被显式地复制为正确的次态。对于立即迁移,类似前面所述,还需要在整个代码中重复。更进一

步,弧是用顺序描述来表述的。以上这三点原因导致平面 VHDL 描述的文字量大约是 SpecCharts 的 4 倍左右。

	概念模型	VHDL			
		SpecCharts	(分级)	(平铺)	Statecharts
描述的特性	程序状态数	42	42	32	80
	连接弧数	40	40	152	135
	控制信号	—	0	84	1
	行数 / 叶子	—	7	27	29
	行数	—	446	1592	963
	字数	—	1733	6740	8088
缺点	无顺序程序结构				×
	无层次		×	×	
	无执行结构		×	×	
	无层次化事件			×	
	无状态迁移结构		×	×	

图 4-13 SpecCharts、VHDL 和 Statecharts 的比较

层次化的 VHDL 描述不需要任何附加的程序状态或弧,但需要增加 84 个控制信号,每个程序状态两个信号,用于在多个进程中控制。读写这些信号,加上对立即迁移的重复和弧的顺序语句表示,也导致四倍 SpecCharts 的文字数量。很明显,描述的行和文字数量越多,系统描述的获取时间、理解时间和错误的出现都会增加。只考虑叶子程序状态,两种 VHDL 描述版本对每个叶状态的描述都需要 SpecCharts 四倍左右的描述量。这种增加降低了叶子状态的可读性,也与用叶结点来使程序功能模块化和便于理解的初衷背道而驰了。

138

Statecharts 缺少程序设计结构的后果在这个例子中也清晰可见。由于叶行为的程序设计结构必须用状态和弧来描述,Statecharts 的描述包含了相当于 SpecCharts 描述的两倍的状态数和三倍的弧数。比起利用顺序程序设计结构而言,用状态和弧来描述程序设计结构是非常单调和不自然的。例如,一个简单的 for loop 就必须使用多个状态和弧来描述。注意,由于 Statecharts 只是图形化的定义,没有定义行和文字。

139

4.7.4 设计质量比较

本实验的目的是说明利用 SpecCharts 设计的质量并不低于从自然语言描述中产生的设计。我们将比较从自然语言系统描述中生成的设计和从 SpecCharts 系统描述生成的设计的晶体管数量差别。

实例的选择还是本章介绍的应答机。将一个自然语言的描述分发给两位设计者,其中一位从描述中直接生成控制器和数据通路;另一位设计者首先利用 SpecCharts 获取系统描述,继而进行自动层次展开,接着从展开后的 SpecCharts 描述中生成数据通路和控制器。以上两种情况,都采用 UC Berkeley 的 KISS 工具将控制器的有限状态机 FSM 描述综合为逻辑电路。

设计的结果如图 4-14 所示。每个人的设计时间几乎一样,都是 30 小时。注意从 SpecCharts 描述中得到的最终设计的晶体管数量并不多于从自然语言描述中得到的数量。事实

上,用 SpecCharts 得到的晶体管数量更少,因为设计中用到的控制状态较少。状态减少的原因如下,采用自然语言系统描述的设计者用有限状态机 FSM 来获取功能,FSM 是描述系统功能唯一精确的模型。为了在脑海中验证电话机功能的正确性,设计者必须维护 FSM 的可读性。这种可读性的要求将阻碍设计者将多个可能的状态合并为一个状态,因为这种合并会提高心算验证的难度。另一方面,SpecCharts 的设计者利用 SpecCharts 描述进行功能验证。当转换为 FSM 以后,FSM 的可读性是不用考虑的。这个转换的过程中进行了状态的合并,因而会得到较小的控制逻辑。

设计特性	从自然语言 得到设计	从 SpecCharts 得到设计
控制晶体管数	3130	2630
数据通路晶体管数	2277	2251
总晶体管数	5407	4881
总引脚数	38	38

图 4-14 设计质量的比较

4.8 结论

140

采用可执行的系统描述比利用自然语言来获取系统功能的描述有更多的优点。它使得设计者在设计流程的早期就明确地考虑尽量精确的功能细节,此时对细节的更改是比较容易的。可执行的系统描述可以利用模拟器进行验证,也可以交由综合工具来自动设计。系统描述也可作为良好的文档供将来使用。这些优点可以导致更短的设计时间和更快的上市日期。

为了创建系统功能的可执行系统描述,设计者必须首先建立能够非常精确且可以理解的用于组织系统的概念模型,必须选择一个支持获取该模型的语言。这一章说明了 PSM 模型和 SpecCharts 语言在获取嵌入式系统的功能描述方面的适用性。

系统设计的可执行描述和软件设计中的结构化程序设计语言,二者的角色有着很强的相似性。最早,软件是在汇编语言基础上开发的。随着程序复杂度的提高,开发的重点开始向具有更高抽象性的结构化程序设计语言的层次上转移,例如 C 语言。这种语言更容易让人理解,从而能够减少功能设计的错误,加快开发的时间。在系统设计方面,以前的设计重点在 FSM 和数据通路组件的层次上。当前,重点已经转移到可执行描述语言如 VHDL 和 Verilog 上,同样是因为这种语言可以极大地提高人对复杂系统的理解。在软件方面,高级语言又演化到处理新的概念模型的层次,例如 C++ 支持面向对象的模型。在系统设计方面,高级语言也将继续发展,如支持 PSM 模型的 SpecCharts。

141

4.9 练习

1. 将图 4-9b 中的号码检查用有限状态机重写。该版本是否更容易理解呢?
2. 假设当前的应答机行为是图 4-8c 中指定的记录消息,且 `tape_rec = '1'`。假设 `power` 变为 '0';如何能使 `tape_rec` 变回 '0' 来中断消息的录制过程?(提示:参考附录 B 的系统关行为)
3. 用 Statecharts 来描述监测器行为。
4. 用 VHDL、SpecCharts 和 Statecharts 分别描述一个具有取指令和执行功能的简单 CPU 的描

述,指令可以自己选择。修改上述每个设计,使得系统可以异步重启。

5. 叙述系统功能描述和它的结构实现之间的主要区别。它们各自的有用方面是哪些?

* 6. 用 SpecCharts 来描述一个不同于本章所介绍的商用应答机的功能。将该功能描述和自然语言的功能描述进行比较。

* 7. 用 SpecCharts 和 Statecharts 来描述一个交通灯控制器。

* 8. 可执行描述和结构化的实现都可以通过模拟来验证功能的正确性。设计一种技术来比较各自模拟的输出,要求该方法能够保证同样的输入集可以得到同样的输出,即使每次输出的时间不同。

** 9. 设计一种方法将自然语言的功能描述转换为一种可执行的系统描述。

** 10. 开发一个自然语言的子集,使得对功能规格的书写可以用精确但是较为自然可读的方式进行。

142

** 11. 高层次综合的任务是将可执行的系统描述转换为寄存器传输结构。然而,编写可执行的系统描述通常只是为了方便可读性,并不一定用于综合。叙述一下可读性和可综合性的系统描述之间的不同,并且在二者之间定义一套用于转换的规则。

143

第 5 章 转换成 VHDL

在前面的两章中,讨论了能最好地支持给定系统模型的系统描述语言所带来的优势。不幸的是,最好的语言并不总是具有最好的支持工具。在本章将说明如何通过将想要的语言转换成另一种具备好的支持工具的语言来解决这个问题。

5.1 引言

使用可执行语言去描述系统所期望的功能意欲达到几个目标。首先,这种语言可以作为系统概念设计(conceptualizing)的媒介,它提供了一种具体实现形式,可以被多个人不断修正和细化,直至完成,从而使定义系统所期望功能的实际智能过程更加简便。其次,当把系统功能用语言表达出来后,我们就能得到具体的系统描述。从这种意义上说,使用可执行语言进行的系统描述提供了一个可以在今后重复调用的文档。最后,可执行语言可以作为各个工具的输入,例如模拟、调试、综合和验证工具。一种理想的语言应该同时实现上面 3 个目标,虽然通常情况下并非如此。举个例子来说,第 3 个目标用标准语言可以很好地实现,因为标准语言本身就具有很多的工具。然而前两个目标,只有支持简明地组织给定系统功能的概念模型(conceptual model)的语言才能实现,详见第 3 章。

145

假设一个建模者已经为给定系统选定了合适的概念模型。理想情况是,使用支持这一模型的标准语言来构造系统模型。不幸的是,通常并不存在一种标准语言可以完全支持所选概念模型的每个特性。例如,程序状态机(program-state machine)可以说是嵌入式系统最好的概念模型,但是大多数标准语言(例如 VHDL)支持的是并发任务模型。系统建模者不可能等待针对他提出的概念模型开发出一种新的标准语言来,所以这种情况要求他采用下面 3 种选择中的一种。

1) 标准语言(standard language):这是应用中最常见的选择。当使用一种已有的标准语言时,必须要克服它的不足,对于每个不能被支持的特性,利用该标准语言中可用的结构进行复杂的组合来描述概念化模型。这个方法有若干优点:首先,尽管该语言不能完全支持所选模型,但是可以扩展该语言已有的工具基础,从而使得模型容易被组合成一个全新的或者已有的框架。同时,学习语言本身非常容易,因为它已经为人熟知了,或者有容易阅读的文档。由于同样的原因,学习这个工具也很容易。另外,不需要实现新的工具,已有工具就有很好的效率。这个方法也有它的缺点,最主要的缺点是要把不能被支持的概念化模型特性强制转换成已有的语言结构。这样模型就只能被隐式描述,结果使得系统描述难以阅读和理解,因而降低概念化的有效性,也降低文档的可读性。从这个意义上讲,选择标准语言与使用正式语言的 3 个目标的前两个相矛盾。

2) 专用语言(application-specific language):在这个方法中,选择与所需要的概念化模型直接对应的语言,即便这种语言可能缺乏大量的支持工具和专门技术。我们称这种语言为专用语言。这种方法的优点是可以保证模型和语言之间有很好的匹配,从而实现使用可执行语言的前两个目标,对概念化和文档有所帮助。这种方法也存在缺点,就是新的工具需要实现和学习。更进一步,这些新工具与已有工具结合,可能不是非常有效和可靠。最后,学习这种语

146

言要花费时间,特别是如果没有可用的好的文档。

3) 前端语言(front-end language):这种方法基本上是结合了上面两种方法的优点。前端语言用于可以转换成标准语言的特殊概念化模型。在这个过程中,可以选择与需要的概念化模型相匹配的语言,使语言满足概念化和文档的目的。然后再将其转换成标准语言,从而利用已有工具。这个方法需要实现一个转换器,但仍旧比实现专用语言所需的模拟或综合工具简单得多。这种方法最主要的缺点是往往需要学习这个前端语言。如果前端语言是由标准语言扩展而来,学习的时间可以缩短。例如,SpecCharts 是 VHDL 的扩展,这意味着熟悉 VHDL 的人只需要几小时就能学会 SpecCharts。图 5-1 举例说明在 VHDL 环境下 SpecCharts 如何作为前端语言使用。

从更一般的层面上看,上面的讨论说明在实际的系统描述和设计环境中语言转换的重要性。更明确地,可以看出语言转换中的关键问题在于模型转换。模型转换就是将系统的一种概念化模型转换成另一种模型。举例来说,如果想把一个用 KISS 语言描述的有限状态机(Finite-state machine, FSM)转换成用 C 语言描述,必须先把 FSM 模型转换成顺序程序模型(sequential program model);这个转换同 KISS 和 C 语言的结构无大关联。另一个例子是把用 C 语言描述的 FSM 转换成

用 SpecCharts 描述。如果仅仅执行语言转换,将每个 C 语言结构转换成相一致的 SpecCharts 结构,SpecCharts 的结果描述只是一个类似于 C 程序的顺序程序。如果用模型转换代替的话,首先从 C 语言中提炼出 FSM 模型,再把 FSM 转换成程序状态机(Program-State Machine, PSM)。SpecCharts 的结果描述是由一系列行为(状态)和弧线组成,不管是人工还是工具都比顺序程序更容易从中辨认出状态迁移的性质。

需要注意的是,如果用不支持所需模型的标准语言进行系统描述,模型转换技术通常必须手动实现。例如,如果必须使用 C 语言描述 FSM,则不得不将 FSM 手动转换成顺序程序。

确定转换技术时有几个问题必须考虑。第一,必须保证输入输出的描述在功能上等价。第二,输出应该是可读的且与输入相关。这两个性质很重要,因为可能需要测试在应用其他工具时的输出,也因为很多公司需要标准语言描述的设计文档。第三,必须保证输出描述在模拟时的效率。第四,输出要服从于综合的需要。

本章的其余部分要讨论将某一模型的特征转换成另一模型的常用技术。当然,全部说起来,有非常多的特征和模型的组合,所以只关注将常见特征转换成并发任务模型(concurrent-tasks model)的技术。选择并发任务模型作为目标模型是因为这种模型可以被 IEEE 标准语言 VHDL 支持,同时被另一种通常使用的标准语言 Verilog 支持。首先讨论将个别特征转换成并发任务模型,然后讨论如何将程序状态机转换成并发任务模型。

5.2 状态迁移

状态迁移允许将功能分解成一组由弧线连接的模式,在 3.2.2 节中已讨论。状态迁移通

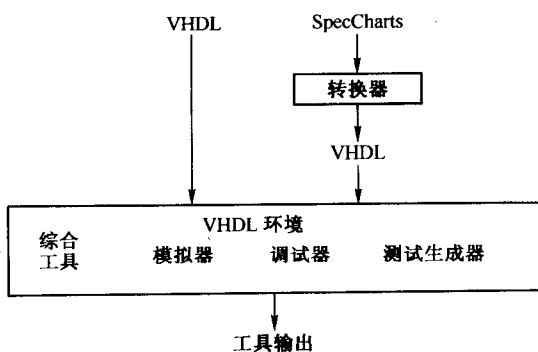
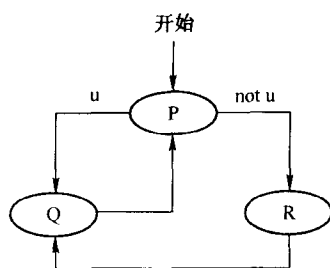


图 5-1 VHDL 环境中 SpecCharts 作为前端语言

常被映像为单任务的顺序描述[DCH91, AWC90, Hal93],因此对状态图的转换包括构造一个状态变量,每个状态动作的一个语句块,以及执行相应语句块的一系列分支。下面举例说明转换方法。

图 5-2a 表示状态 P 、 Q 、 R 之间的一组状态迁移。这些迁移可以转换为等价的一组顺序语句,如图 5-2b 所示。在转换过程中,状态用一个变量表示,它有三个可取值: P 、 Q 或 R ,初始值为 P ;构造一个无限循环来描述状态间的无穷迁移;在循环中,加入了一个 case 语句,由当前状态值决定执行 P 、 Q 或 R ;后面跟由语句描述的动作,依照弧线上的条件,从当前状态迁移到下一个适当的状态。



a) 原始状态机

```

type state_type is (P, Q, R);
variable state : state_type := P;

loop
  case (state) is
    when P =>
      <actions for P>
      if (u) then
        state := Q;
      else if (not u) then
        state := R;
      end if;
    when Q =>
      <actions for Q>
      state := P;
    when R =>
      <actions for R>
      state := Q;
  end case;
end loop;
  
```

b) VHDL 顺序语句

图 5-2 状态机转换

为了分析状态迁移是如何用语句描述的,我们来看执行过程:初始状态下,状态值为 P ;执行与 P 的动作相对应的第一个 case 分支;接下来检查弧线上的条件 u ,如果 u 为真,状态转换为 Q ;反之,状态转换为 R ;于是到达 case 语句末尾,然后回到循环的开始;如果状态值为 Q ,则执行与 Q 的动作相对应的第二个 case 语句分支,这时状态又变成 P ,然后重复整个过程。

149

上述的转换技术适用于单状态机。然而,概念化模型通常描述为并行执行的通信状态机(communicating state-machine)。可以很容易地对这些通信状态机进行操作,把每个状态机映像到它自己的任务之上,并且通过全局变量进行通信(例如 VHDL 中的信号)。

其他的概念化模型描述层次化状态机(hierarchical state-machine),可以使用多种方法进行转换。例如,一种技术是将层次化状态机转换成顺序语句,保留原有层次;而另一种技术则将层次平面化。前一种技术,即层次保留技术,采用下述的方法来修改上面介绍的描述转换技术。回顾前面所讲的,是用 case 语的一个分支表示状态的动作。然而在层次化状态机中,这个分支可以描述为另一个状态机,包括新的状态变量、循环和 case 语句。考虑图 5-3a 与图 5-2a 是相同的,但 P 现在是包括另一个状态机的层次化状态。顺序语句描述的转换也与前面在图 5-2b 中所示的那样,所不同的是: P 的动作由执行另一个状态机的过程呼叫所取代。图 5-3b 表示出了执行 P 的过程:当控制流到达 P 的完成点时,过程返回,说明图 5-2b 中 P 的行为已经完成。

150

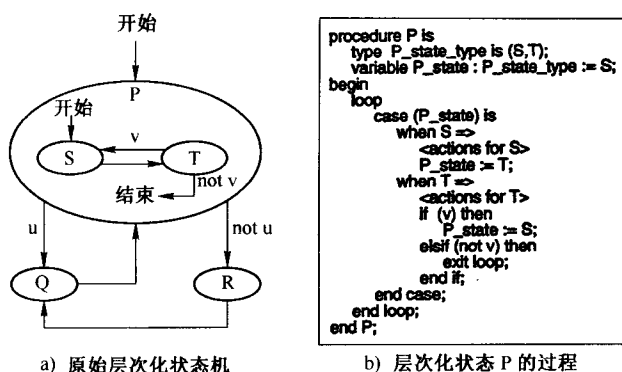


图 5-3 层次保留状态机转换

另一种方法,在层次平面化技术中,层次化状态机平面化,只剩下叶子状态。然后把平面化的状态机转换成如图 5-2 中所示的顺序语句描述。例如,可以将图 5-3a 中的状态机平面化,成为图 5-4a 所示的新状态机。然后就能将其转换成图 5-4b 中的语句描述。

这几种技术各有优势。层次保留技术的优点是可以很容易地将顺序语句描述同原来的状态机对应起来,因为层次是相同的,而且弧的数目不变。另一方面,而层次平面化技术的优点在于根据顺序语句综合所得到的硬件更加简单。

151

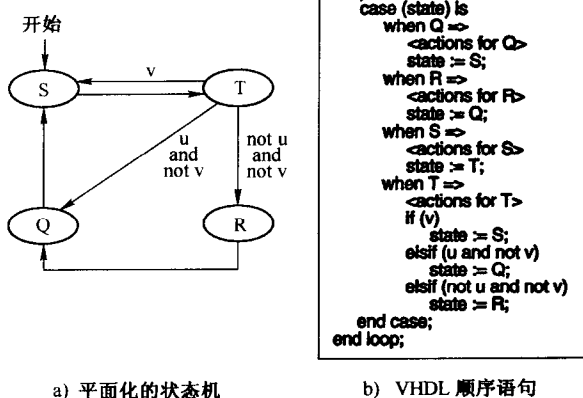


图 5-4 层次平面化的状态机转换

5.3 消息传递通信

使用消息传递结构(message-passing construct)是为了保证并发性行为的通信的抽象且容易理解,在 3.2.6 节中已经讨论过。在消息传递通信中,数据通过抽象的通信通道发送和接收。这种消息传递结构可以映像到没有这种结构的语言上,例如 VHDL,用共享变量来代替通道,而通过这些变量传递数据的显式动作来实现“发送”和“接收”。

5.3.1 阻塞式消息传递

在阻塞式消息传递(blocking message passing)中,消息发送器必须等到消息被接收后才能

继续工作。在图 5-5a 中的例子说明了这种消息传递的转换。通道 *chan* 用于传递整型数据。进程 *P1* 通过这个通道由变量 *msg* 发送整型数据,进程 *P2* 接收这些数据到变量 *m*。图 5-5b 为转换后的语句描述。用全局变量即 VHDL 中的信号来代替 *chan* 的声名,并且添加了两个新的全局信号:*chan_req*,用来说明 *chan* 中有可用消息;*chan_ack*,用来说明 *chan* 中的消息已经被读出。同样地,用对 *chan* 的写操作和对 *chan_req* 的赋值来代替 *P1* 中的发送操作。最后,用等待 *chan_req* 被赋值和对 *chan* 的读操作代替 *P2* 中的接收操作。这样,*P1* 通过写入 *chan* 和初始化与 *P2* 的握手来发送数据。握手完成后,*P1* 知道数据已被接收,于是继续执行。

152

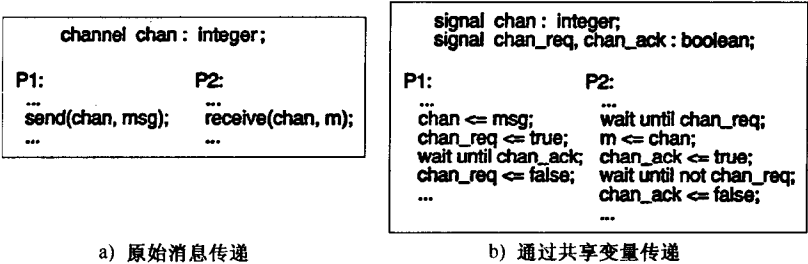


图 5-5 阻塞式消息传递转换

注意,在前面的例子中,数据传递是发送器发起的。如果是由接收器发起的,传递方式是类似的,所不同的是接收器要发出请求,并且要等待发送器的确认,指出消息是可用的。接收器读出消息后取消请求。

上述转换技术的困难之处在于,虽然生成了功能正确的代码,但代码往往被发送和接收时所需的语句细节搞得很混乱,不具备可读性。然而,我们可以很容易地把发送和接收的细节封装在某些过程中,从而提高代码的可读性。如图 5-6a 中那样,将所有与通道有关的信号组合为一个记录结构,可以进一步提高代码的可读性。在这个例子中,声明了一个记录类型 *integer_channel*,包括三个域:一个整型域表示消息,两个布尔类型域表示请求和确认。进而,定义了一个发送过程 *send*,它有两个参数,分别表示通道和其他消息。接收过程 *receive* 的声名与之类似。有了这两个过程和记录类型,只需要正确声明一个记录类型的通道,并利用发送/接收例程,如图 5-6b 中所示。值得注意的是,在这种通信方式下,用包含显式消息传递结构的语言接近于抽象描述(比较图 5-5a 和图 5-6b)。

153

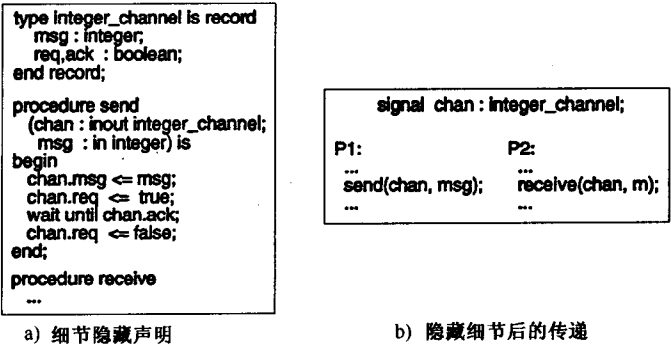


图 5-6 消息传递转换

如果目标语言允许过程名的重用,像 VHDL,则可以声明多个名为 *send* 和 *receive* 的过程,每个程序具有不同的通道和消息类型。否则,可以给每个发送/接收过程起一个唯一的名字,用来区别所传递的数据类型,例如 *send_integer* 和 *send_character*。

5.3.2 非阻塞式消息传递

154 非阻塞式消息传递(non-blocking message passing)与阻塞式消息传递的区别在于:非阻塞式消息传递的发送器不用等待(阻塞)到消息被接收后再继续执行。其结果造成在某一特定时间,通道内可能会有不止一条已经发送但尚未接收的消息。在这种情况下,通道按一个隐含队列工作:每次发送时将一条消息添加到队列中,每次接收时从队列中删除一条消息。要实现非阻塞式消息传递的转换,应修改我们的转换技术以明确地描述这个队列。

对于每个通道,用图 5-7 中所示的进程模拟队列,声明一个描述队列的变量。现在假设已经定义了一个队列数据类型及其访问例程(例如添加、删除)。从 *P1* 到 *P2* 的非阻塞式数据发送可以替换为从 *P1* 到队列进程的阻塞式发送和从队列进程到 *P2* 的阻塞式接收。当 *P1* 请求发送时,队列进程接收消息并将其添加到队列中。当 *P2* 请求接收时,队列进程从队列中删除一条消息并将其发送。尽管发送和接收被转换成阻塞的,阻塞并不会使传递发生时延。没有时延是因为队列进程对发送和接收请求的响应非常迅速,因为它除了响应请求外别无他用。形成对比的是,图 5-5

```
process
variable msg_queue : integer_queue_type(queue_size);
begin

wait until chan.send_req or chan.rec_req;

if (chan.send_req) then
QueueAdd(msg_queue, chan.send_msg);
chan.send_ack <= true;
... -- finish handshake
end if;

if (chan.rec_req) then
QueueDelete(msg_queue, chan.rec_msg);
chan.rec_ack <= true;
... -- finish handshake
end if;

end;
```

图 5-7 非阻塞式通道的队列进程

155 中所示的阻塞式传递中不存在这种迅速的响应,因为 *P2* 还有很多其他的动作,因而也许还没有准备好对请求做出迅速响应。与队列进程进行通信只需要很小的握手时延,这是可以忽略的,也就是说发送器的总体效果与非阻塞式情况相同。

注意每个通道需要两组消息、请求和确认信号,一组用于发送器和队列进程之间,另一组用于队列进程和接收器之间。

同时也应注意的是:队列的分配既可以是静态的也可以是动态的。如果使用静态分配,队列空间必须足够大,要能够处理队列中可能会出现的最大数目的消息。尽管隐含队列应该是无穷大的,但对于特定系统仍可以用[AB91]中的技术来确定其队列大小最大值。这是很重要的,因为如果队列有固定的大小,它就有可能被充满,这时发送请求就会被阻塞,直到接收操作从队列中取走数据。

[LGR92]中提出了一种类似的建模非阻塞式消息传递的方法,其不同点是它为每个过程构造一个发送接收消息的进程,而不是对每个通道构造一个进程。

综上所述,把阻塞式和非阻塞式消息传递结构转换成不支持此结构的语言时仍可保持通信的抽象性,因为可以很容易地将细节封装在过程中。

5.4 并发

很多模型都可以描述并发行为。其中有些模型允许数据流描述,每种操作同其他操作并

行转换数据。其他的模型允许顺序控制线程分叉成若干线程,然后再合并成一个线程。下面说明如何将这两种不同的并发类型转换成并发任务模型。

5.4.1 数据流

在数据流描述中,有一组操作将输入数据流转换成输出数据流,如 3-1 节所讨论的。将数据流图转换成顺序语句有两种技术,一种技术对每个数据流图使用一个进程(数据流图进程技术),另一种技术对每个数据流操作使用一个进程(操作进程技术)。

156

在数据流图进程(one-process-per-graph)技术中,每个输出通过算术表达式描述为输入的函数。所有输出在同一进程中计算,当任意输入发生变化时,都要重新计算所有输出。举例说明,考虑图 5-8a 中的数据流图。输入 I 和 J 相加,生成输出 O 。相加的结果作为输入连同输入 K 一起进行操作 Op ,生成输出 P 。图 5-8b 为该这个数据流图转换成的进程,计算中间值 sum 并将它赋值给 O ,然后将 $Op(sum, K)$ 的结果赋值给 P , Op 写成一个函数。当 I 、 J 或 K 发生变化,该进程重复以上计算过程。

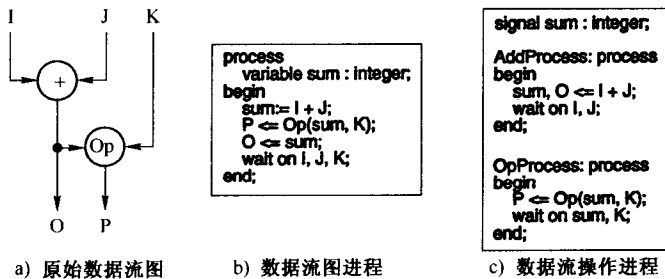


图 5-8 数据流转换

另一种选择是使用操作进程(one-process-per-operation)技术,每个操作都描述为一个进程,操作间的数据用全局信号表示。每个进程监控输入数据的变化,检测到变化后就重新计算新的输出数据。图 5-8c 是这种技术的一个例子。过程 AddProcess 计算 $I + J$,将结果赋值给全局信号 sum 和输出 O 。当 I 或 J 变化时,重复此过程。同时,另一个过程 OpProcess 计算 $Op(sum, K)$ 并将结果赋值给输出 P 。当 sum 或 K 发生变化,重复此过程。

157

这两种技术可以结合起来,对于给定的数据流图,每个子图用一个进程表示。进程等待子图的输入变化,然后得到子图的输出。如果把每个操作看做一个子图,那么操作进程技术就化为数据流图进程技术的一种特殊情况。

可以扩展这些转换技术,对层次化数据流图进行操作。层次化数据流图中的操作可以定义成另一个数据流图。如同对层次化状态机的处理方法,在转换时可以保留层次或者层次平面化。在每个数据流进程技术中,通过详细描述每层操作的子图保留层次。在操作进程技术中,每层操作用一个过程描述,这个进程分成若干个与子图中的操作相对应的过程。与状态机相同,保留层次的优点在于数据流图和生成的进程间存在简单的对应关系。这样做的缺点是缺乏硬件高效性。

另一种层次化操作技术在[TLK90]中有说明。这项技术实质上与操作进程技术相同,扩展之处在于把每个进程封装成一个结构元素。这种封装可以很好地支持层次化数据流图,由于每个操作用一个元素表示,多层操作就可以用一组互联元素表示出。

还可以扩展这些数据流转换技术,处理不明操作(uninterpreted operation)。不明操作没有

对功能的详细解释。在这种情况下,当输入有效时生成输出,但是输出的值是未知的。这种输出称为标记(token)。对这种操作的转换技术与上面说明的操作进程技术类似:同样声明全局信号来表示操作之间的数据,但是这些信号是简单的布尔类型,用来说明标记是否存在。每个进程等待它的每个输入都有了一个标记,然后为它的各个输出生成一个标记。这种处理不明操作的技术在[HAWW88]中有所说明。

最后,还可以扩展这些转换技术来处理允许数据排队(实际数据或者标记)的数据流模型。在转换过程中,队列可以使用 5.3.2 节中的技术显式表示出来。

5.4.2 分叉

分叉(fork)是在一个控制线程中激活若干并发线程的点。在某些情况下,这些活动线程是简单的语句;其他情况下,它们是复合任务。下面说明如何处理以下两种类型的分叉。

1. 语句级(Statement-level)

语句级分叉出现在单一控制线程并行执行若干语句时。完成这些语句后,线程继续执行后面的语句。通过插入临时变量能把语句级分叉转换成顺序语句描述。临时变量的功能是:保证并行语句中所有变量的值在所得到的顺序语句中保持不变,只有在全部初始并行语句都执行后才能更新。

例如,图 5-9a 中是一个语句级分叉结构。*Statement1* 执行后, $x := y$ 和 $y := x$ 并行执行,实现 x 和 y 的交换。交换后,继续执行 *Statement2*。图 5-9b 是将同一个分叉转换成一种不存在并行语句结构的语言。在转换的过程中,将并行语句的每个写变量操作替换成写入临时变量。于是,写入 x 被替换成写入 x_temp 。在所有语句的最后,加入了用临时变量的值更新每个变量值的语句。因而 x 被设置为 x_temp 的值。注意,如果一个变量,例如 y ,在并行语句中写入之后不被读出,就无须引入临时变量,因为立即更新变量不会影响最终结果。

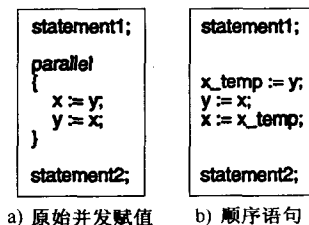


图 5-9 并发赋值转换

将并行语句映像到顺序语句的过程非常有用,特别是基于软件编译的目的,将之转换成标准的顺序程序设计语言的情况下。

2. 任务级(Task-level)

任务级分叉出现在单一控制线程并行激活多个任务时。完成一个或多个这样的任务后,该线程继续执行。通过引入一组全局信号能把任务级分叉转换成一组并发任务,全局信号的作用是激活分叉中的任务,并且说明任务的完成情况。

例如,图 5-10a 中是一个任务级分叉。*Statement1* 执行后,程序段 $P1$ 和 $P2$ 并行执行。当两个程序段都完成后,*Statement2* 继续执行。注意到这个例子与上个例子唯一的区别在于并行执行的行为的规模。图 5-10b 表示将分叉转换成一组并行进程,每个分叉的程序段对应一个进程: $P1_process$ 等待全局信号 $fork$ 被赋值,接着执行 $P1_process$, 且赋值给 $P1_done$ 。 $P2_process$ 的定义相似。代替分叉的是一个赋值给 $fork$ 的语句,以及等待 $P1_done$ 和 $P2_done$ 被赋值的语句。

为了说明变化后是如何执行分叉的,考虑下面的执行过程:首先执行 *Statement1*。然后赋值给 $fork$,激活 $P1_process$ 和 $P2_process$ 。 $P1_process$ 调用程序段 $P1$, $P1$ 返回后,赋值

给 *P1_done*。*P2_process* 的执行过程相类似。经过一段时间, *P1_done* 和 *P2_done* 都被赋值, 继续执行 *Statement2*。其他进程等待 *fork* 被再赋值。

[NVG91b, JPA91, MW90] 中详细阐明了类似的分叉操作技术, 主要的区别在于控制信号的数目。

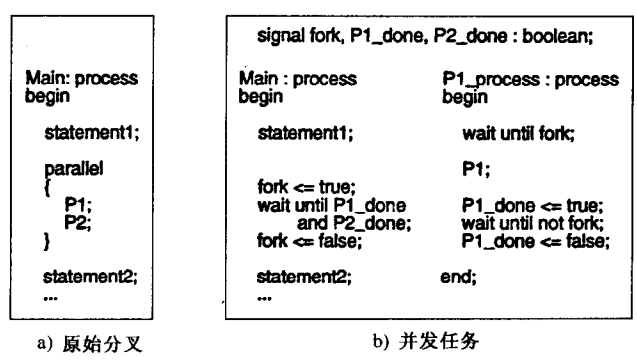


图 5-10 分叉转换

5.5 异常处理

通常, 由外部事件导致的异常要求迅速终止正在执行的操作。插入对事件的检查, 检测到事件后跳转到程序末尾, 就可以把描述这种异常的结构转换成一组顺序描述。这种对一个事件重复检查称为转态 (polling)。

例如, 图 5-11a 给出了一组名为 *T* 的语句, 事件 *e* 发生后立即停止。停止后执行语句 *S*。图 5-11b 表示出了转换成没有异常结构的语言后对异常的操作。 *T* 中的每个语句后都有一个检查事件的 *if* 语句。如果事件 *e* 发生, 通过一条 *goto* 语句跳转到 *S* 的开头, 忽略 *T* 中剩余的部分。如果事件没有发生, *T* 正常执行。

需要注意的是, 上述方法的基础是假设目标语言支持 *goto* 语句。然而, 为了鼓励结构化程序环境, 大多数的程序设计语言不支持 *goto* 语句。虽然如此, 大多数语言还是支持从循环中的任意位置跳出, 适合进行异常处理。图 5-11c 举例说明如何利用跳出循环的语句代替 *goto* 语句进行异常处理。将 *T* 放在一个循环中, 如果检测到了事件 *e*, 就跳出这个循环, 接着执行 *S*; 如果没有检测到事件, 要在执行完 *T* 后再退出循环。换句话说, 永远不会返回到 *T* 循环的开头。

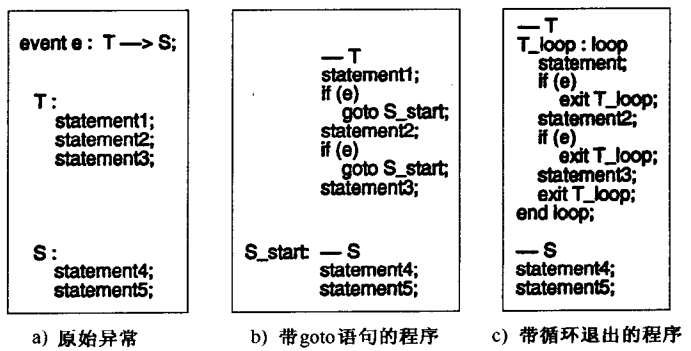


图 5-11 异常转换

这种转换技术还适用于结合了时间模型(timing model)的语言。在这种语言中,只有专门的语句才使得时间推移,例如 VHDL 中的 wait 语句。因为只有时间推移时事件才能发生,所以只需要在时间推移语句(time-advancing statement)后进行事件检查。

上述技术在实现正确功能的同时,也存在着缺点,这是因为重复的事件检查使结果描述显得很混乱,原始功能变得难于理解。有一种常用工具可以减少混乱,使结果语句容易阅读。这个技术把原始语句集合分成若干子集,每个子集看成一个状态,当前状态用弧线连接到下一个状态,与接着要执行的语句集连接起来。在这个方法中,异常处理就是给状态机的每一个状态添加一条指向最终状态的弧线,只有当异常发生时才沿此弧线转换。这个状态机可以按照 5.2 节中的方法进行转换。例如,图 5-12a 给出了 T 转换成的 FSM,包括添加的异常处理的弧线。可以将这个 FSM 变回图 5-12b 中的顺序程序。因为每个状态之后都会检查事件 e,所以可以仅在循环条件中进行检查。这样,任意时刻如果 e 发生了,不会转移到下一个状态,而是接着执行 S。注意这里只有一条检查事件 e 的语句,与图 5-11c 中的多条语句不同。

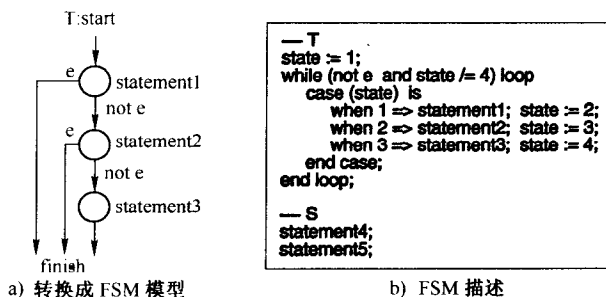


图 5-12 异常转换

上面的例子很清楚地说明了不同语言的转换与不同模型的转换两种转换之间的区别。顺序程序模型中的异常处理首先被转换成 FSM 模型中的结构,然后又变回顺序程序模型。图 5-11c 中的程序是由原始顺序程序直接转换而来,图 5-12b 中的顺序程序结果比图 5-11c 中的程序更加结构化和易读。

5.6 从程序状态机到任务

5.6.1 概述

本章前面各节介绍了一组各种模型性质转换的技术。现在可以考虑如何借由对这些技术的处理将整个程序状态机概念化模型 PSM 转换成并发任务模型。这种转换可以用来实现诸如 SpecCharts 到 VHDL 的语言转换。

PSM 模型能支持并发任务模型不能支持的三个特性:状态迁移、任务级分叉和异常处理。一种直接的转换方案是利用本章介绍的技术对每个特性分别转换。但是如果能对前面的技术稍加修改,就能在保证并发任务与原始 PSM 功能等价的同时,大幅度提高生成描述的可读性和关联性。

首先,将每个程序状态映像到它自己专有的进程上,而不是将顺序程序子状态映像到 case 分支语句中的动作。因为如果每个程序状态都有一个进程,PSM 就能很容易地同这些进程对应起来。其次,使用一个信号来说明某个程序状态的进程应该终止,而不是显式地对进程中所

有语句进行转态。新的信号可以是布尔类型的,当任意相关的异常发生时其值为真。使用这个信号可以使生成进程的表达式更加简单,更加易读。

根据这些修改后的技术,现在可以引出一种将 PSM 转换成并行任务的新的方法。的确,已经有人研究了将 PSM 的某些特性转换成并发任务的方法[DCH91, AWC90, JPA91, MW90],但是没有人用统一的方法研究过 PSM 的全部特性,这正是我们的目标。

在这种转换方法中,总是认为每个程序状态有挂起(*inactive*)、执行(*executing*)或者完成(*complete*)三种状态之一。PSM 模型中的每个程序状态能映射到具有这三个不同部分的进程上。在“执行”状态中,表示复合程序状态的进程会设置和清除控制信号,以便激活或者挂起程序子状态对应的进程。如果目标语言在声明域内支持模块结构,则可以使用嵌套结构来保留 PSM 的层次。

164

5.6.2 算法

算法 5.6.1 是利用上述概念将 SpecCharts 转换成等价的 VHDL 描述的递归算法。详细的算法参见[NVG91b]。在 SpecCharts 中,程序状态称为行为(*behavior*),在 VHDL 中,任务用进程描述。算法的输入是 SpecCharts 行为,输出是 VHDL 描述。算法使用子程序 *CreateCompletionHandshake(B)* 生成一组描述:首先,通过给信号 *B_complete* 赋值,指出行为 *B* 已经完成;然后等待父行为通过清除信号 *B_active* 的值使 *B* 挂起;最后清除 *B_complete*。子程序 *CreateWaitOnArcs(arcs)* 生成 *wait* 等待语句,用来判断是否要沿着弧线发生迁移。当子行为 *S* 已经完成,并且满足弧线上的完成迁移(*transition-on-completion*)条件,或者子行为 *S* 正在活动,并且满足弧线上的立即迁移(*transition-immediately*)条件,就会发生迁移。子程序 *CreateArcsIf(arcs)* 为每个弧线分支生成 *if-then-else* 语句。每个分支的描述,终止当前的子行为,同时激活弧线的目的子行为。子程序 *InsertPolling(stmts, active_sig)* 在程序段 *stmts* 中插入转态代码,如果 *active_sig* 被赋值则跳转到程序末尾,如 5.5 节中所讨论的。子程序 *CreateSignalShutoffs(stmts)* 停止 *stmts* 中全部信号的驱动。这个操作是很必要的,因为非活动行为的进程必须完全被忽略,所以不应再驱动信号值。VHDL 信号驱动器的更详细的说明参见[IEE88]。最后,子程序 *Append(l, m)* 可以把表 *m* 接到表 *l* 末尾。

从当前层次的顶端(根)行为开始,算法按深度优先的顺序遍历行为,对每个访问的行为输出 VHDL。每个行为的 VHDL 代码封装在一个模块中,使得嵌套模块保留了 SpecCharts 的层次和声明的正确作用域。

165

算法 5.6.1: BehaviorToVhdl(B)

```

output start of block labeled B_block, with B.declarations
if IsCompositeBehavior(B) then
  for each S in B.subbehaviors loop
    output boolean signal declarations S_active, S_complete
    BehaviorToVhdl(S) /* 递归调用 */
  end loop
end if

/* 挂起部分 */
stmts = NULL

```

```

Append(stmts, "wait until B_active;" )

/* 执行部分 */
if IsCompositeBehavior(B) then
    Append(stmts, "loop")
    Append(stmts, CreateWaitOnArcs(B.arcs))

    Append(stmts, CreateArcsIf(B.arcs))

    Append(stmts, "end loop;")
elseif IsLeafBehavior(B)
    Append(stmts, B.statements)
end if

/* 完成部分 */
Append(stmts, CreateCompletionHandshake(B))

InsertPolling(stmts, B_active)
Append(stmts, CreateSignalShutoffs(stmts))
output process with label B and statements stmts
output end of B_block

```

166

VHDL 进程包含在与每个行为相关的 VHDL 模块中,包括三个部分:

- 1) 挂起(Inactive):在这个部分,行为等待被激活(与父行为对应的进程给控制信号赋值)。
 - 2) 执行(Executing):这个部分的复合行为通过控制信号激活或挂起相应的子行为。叶子行为执行其 VHDL 代码。
 - 3) 完成(Complete):通过控制信号指定已完成,等待父行为清除控制信号值后挂起。
- 注意算法可以通过控制信号 *B_active* 实现挂起,因为在这种情况下进程能跳转到其末尾,使得进程重新进入挂起部分。

本节介绍的转换算法具有若干优点。首先, VHDL 与 SpecCharts 在任何情况下的功能都是等价的,而不像前面的方法那样,仅仅在一个子集内等价。其次,算法在 VHDL 输出中保留了 SpecCharts 的行为层次,而且 SpecCharts 行为和 VHDL 进程是一一对应的, SpecCharts 和 VHDL 之间存在简单的关联。最后一点,算法的递归形式很容易实现。

5.6.3 时间调整

我们说明的转换方法存在一个问题,可能导致不正确的模拟。SpecCharts 允许信号在一个无限小的时间单位(称为 *delta*)后更新。这些信号被转换成 VHDL 信号。同时,转换方法还引入了新的用于进程控制的信号。现在假设因为某个事件,行为 *B* 需要被挂起。*B* 的父行为会检测到这个事件并给控制信号赋值,使 *B* 终止它的动作。然而,进程控制信号只有在 *delta* 时间后才能更新,在这期间, SpecCharts 可能在不该更新的时候更新了。概括地说,进程控制信号更新时需要的 *delta* 时间可能会干扰到 SpecCharts 信号更新时需要的 *delta* 时间。

167

简单的解决方法是将 SpecCharts 信号更新所需的时间调整为比控制信号更新所需的时间

更长的时间单位。例如,可以将 SpecCharts 信号更新所需的时间调整为飞秒(femtosecond)的时间级别;类似地,可以把一个时间调整到比另一个时间高一个数量级。在这种情况下,模拟输出的时间也需要调整。方法细节参见[VG91]。

需要注意的是,同样的信号干扰问题在任何引入了控制信号的转换方法中都可能出现,前面涉及的大多数方法都有可能。在每种方法中都可以这样进行时间调整。

5.6.4 综合

用这种转换方法生成 VHDL 还有另一个潜在的问题:当使用 VHDL 综合工具的时候,可能会带来硬件上的低效,主要是因为当前的综合工具假设,实现每个 VHDL 过程需要一个控制器和一个数据通路。因为生成的 VHDL 每个行为只包含一个进程,根据 VHDL 进行综合的结果可能会生成额外的控制器和低效的数据通路。解决这个问题的简单方法就是在转换前将多层次化行为展平,成为单个叶行为的顺序语句。这种展平化易于自动实现。

5.7 结论和发展方向

前端语言是任何系统描述环境的基本组成部分,因为它使我们可以用自然易理解的方式描述多种功能。在实际应用中,为了利用已有的工具和技术,前端语言必须转换成标准语言。

本章介绍了将一些前端语言常见的性质转换成标准语言 VHDL 的技术。类似的技术可以用来将其他的前端语言转换成多种标准语言。将这些技术结合起来就是一个转换器,这使转换器实现起来比模拟器这种语言支持工具简单得多。正如前面章节所提到的,使用这个转换器支持前端语言可以充分地减少描述时间和减少功能性错误。

今后的工作包括开发新的前端语言,为多种系统开发转换方法。还有支持前端语言开发的工具,例如调试器,与已有的标准语言的支持工具衔接。

5.8 练习

1. 考虑如下顺序语句描述的 VHDL 过程:

```
process P
begin
    wait on start for 100 ns;
    for l in 1 to n loop
        wait on input;
        sum := sum + input;
    end loop;
    wait;
end process;
```

假设当条件(interrupt = '1')为真时,想要跳转到最后的 wait 语句,暂时挂起进程。需要如何修改上面的进程才能进行这个异常处理?

2. 重复进行上面的进程 P 中的计算,利用一个其动作仅与状态间迁移相关的有限状态机。
3. 考虑如下的一组包含 fork-join 的语句:

168

169

```
P();  
fork  
  Q();  
  R();  
join  
S();
```

假设 $P()$ 、 $Q()$ 、 $R()$ 和 $S()$ 表示一组顺序语句。用只有单层进程的语言(像 VHDL)表示上面的 fork-join 模型。

4. 说明使用顺序语句描述有限状态机的 5 种不同方法。
5. 编写一个将有限状态机转换成一组顺序语句描述的算法。
6. 编写一个将层次化有限状态机变成只有叶子状态的算法。在处理层次化 FSM 前经常需要这样做。忽略并发子状态。
7. 编写一个算法将层次化程序状态机展平成为只包含叶状态的程序状态机。需要处理 TI 和 TOC 弧。忽略并发程序子状态。
- * 8. 简要说明一种将一组顺序语句描述转换成一组等价的数据流描述的方法。(提示:参考 [CG93] 中的分配决策图。)
- * 9. 简要说明一种算法,将一条任意复杂的 VHDL 的 wait 语句转换成一组语句,其中只有一条很弱的 wait 语句,例如,wait until CLK = '1'。这个算法是 VHDL 到 C 的转换器的基本组成部分。
- ** 10. 简要说明一种从顺序描述中提取出有限状态机的方法。该方法应能支持 FSM 的多种表示方式。

第6章 系统划分

在前四章里面,我们介绍了系统功能描述的设计,对其中的模型、语言和转化等关键问题进行了讨论。在获得系统功能描述后,设计者需要关注的就是系统设计本身。系统设计中一个最重要的和最具有挑战性的任务就是对系统功能进行划分,即在满足各种设计约束的前提下,将系统所要执行的功能分到各个组件上。本章将讨论系统划分中的一些基本问题和相关技术。

6.1 引言

一个系统中含有一系列组件,如标准处理器、微控制器、通用或专用集成电路、存储器、总线等,通过这些组件的互联通信,共同实现整个系统的功能。为了获得这种功能的实现,设计者必须解决如下两个问题:一是选择一系列系统组件,即分配(allocation);二是将系统功能分到这些组件上,即划分(partition)。分配和划分必须经过选择,使得最后的实现能满足设计成本、性能、芯片面积和功耗等设计约束。

171

本章将对系统划分进行讨论。首先对结构划分和功能划分进行对比,并针对一个可执行描述来详细讨论功能划分。接着对划分中的主要问题及基本算法进行定义,介绍硬件组件功能划分中的一些已公布技术。然后对软硬件功能划分的算法和技术进行讨论,硬件组件由设计结构实现,而软件组件由软件编译所得。在本章最后,将给出一个在可执行描述划分中广泛应用的例子,通过这个例子,读者可以更好地掌握这一章的内容。

6.2 结构划分和功能划分

系统划分有两种不同的方法,即结构划分和功能划分。前者先从结构上来实现目标系统,然后对其进行划分;而后者则先对系统功能进行划分,然后才具体实现。下面将详细分析两种划分方法的特性及各自的优点。为简便起见,假设在进行划分之前,系统组件已经经过了手工分配。

6.2.1 结构划分

结构划分先从结构上来实现目标系统,然后对其进行划分。这里的结构指硬件对象之间的互联,其中硬件对象可以是逻辑门、触发器、寄存器传输级元件如计数器或寄存器等,也可以是浮点乘法器、傅里叶变换器等复杂的计算单元。接着,我们划分结构。划分就是把这些硬件对象分为不同的组,其中每组都代表一个系统组件。

结构划分应用很广泛,这在一定程度上依赖于其自身的优点。首先,通过结构划分,能够直接获得芯片面积和引脚的估算值,一个组的面积可以由这个组所包含元件的总面积来估算,而引脚数可以根据该组和其他组之间的连线数来估算。另外,结构划分问题可以映射为图的划分问题,且对图的划分问题有一套成熟的理论、复杂的算法和相应的工具来进行处理。结构划分是这样转化为图的划分问题的:首先定义一个图,它包含有若干个顶点和若干条边,其中每条边连接着顶点的一个子集。每个顶点和每条边都赋有一个值,称为权重。定义结群(clus-

172

ter) p 为顶点的一个子集。 p 的总权重(total weight)为其中所有顶点权重的总和; p 的割值(cutsize)为将其内部顶点与外部顶点进行连接的各条边的权重之总和。将系统结构中的每个硬件对象映射为图的一个顶点,并用该对象的面积作为权重;再将其中每条互连线映射为图的一条边,以互连接的线数作为该边的权重。通过这样的映射就可以用对应的图中一个结群的权重估算结构组面积,用对应该结群的割值估算该结构组的外部引脚数。

对于很多系统,结构划分都能获得较好的结果,这是它得到广泛应用的另一个原因。在过去,系统组件的数量与需要划分的硬件对象相比并不是太多,而且硬件对象的数量也不是特别多,这种事实使得该方法容易获得较好的结果。但是,需要注意的是:随着近年来系统组件和实现系统所需硬件对象的数量不断增长,结构划分的能力受到了较大影响。

总的说来,结构划分存在以下三个缺点:

1) **面积/性能的折中比较困难**。在实现系统结构的时候,由于后续的划分步骤可能使先前的决定无效,因此很难在面积和性能之间取得较好的折中,这个问题在片间通信和硬件中尤为突出。如下面这种普遍的情况所示:设计者希望获得一个满足性能约束的最小的系统。一种方案是两个行为共享一个加法器,假定共享加法器会减小芯片面积,并且不会影响性能。但是划分的结果不一定将实现这两个行为的对象分到同一个芯片上。为了获得加法器的输入数据,就必须有芯片间的数据传输,这势必影响系统性能,因此共享加法器这种方案并不好。另一种设计方案是每个行为都使用各自的加法器。但如果划分结果将实现这两个行为的对象都划分到同一个芯片上,那么一个加法器就能满足使用。这个例子是对一个加法器做代价和性能间的决策,而在实际设计的系统结构中需要成百上千个类似决策。通过上面的简单例子,可以看出在划分之前做这些决策可能会导致很差的结果。

一种解决方案是在划分后对结构进行调整。然而却不容易对结构做较大的调整。这是因为一个简单行为会映射到多个对象上,并且多个行为可能共享同一个对象。改变一个对象可能会影响到其他行为的实现部分,并且无法估计这种影响会有多大。因此通常只能对结构做一些较小的调整,如引入冗余门,或者锁存器一端的门移到另一端。对于软件情况也类似:在产生汇编代码后,仅仅能对性能做一些较小的优化,如使用寄存器来避免访问存储器,而类似于把代码移植到多处理器系统上这样大规模的改变则很难实现。对于硬件和软件的设计,具有实质性的调整都需要高层的功能和时序等信息,而这些信息很难从底层的结构和汇编代码中提取出来。

2) **对象数量巨大**。由于系统面积的增大,实现系统所需对象的数量也不断增长。这种巨大的对象数量,一方面使得划分算法的结果变差;另一方面也增大了设计者在划分过程中指导划分的难度,同时划分结果也更难被设计者所理解。

3) **全硬件方案**。结构划分的第三个缺点是它只能应用于硬件设计。如果将处理器或微控制器作为系统组件来进行分配,那么部分功能将会被编译成软件。而在结构划分中,所有功能都首先被转换为结构,因此这种软件解决方案将被忽略。

随着系统越来越复杂,系统组件数量越来越多,结构划分的这些缺点显得更加突出。

6.2.2 功能划分

功能划分首先将系统的功能分为若干个不可再分的块,这些块称为功能对象(functional object)。接下来将这些功能对象划分到系统组件上,再通过软件或硬件来实现每个组件的这些功能。功能划分和结构划分相比,主要有以下几个优点。

1) **面积/性能的折中**。这是功能划分的一个最主要的优点。在功能划分的后续结构实现步骤中,由于有了所有划分的信息,能够在面积和性能间获得较好的折中。这些信息还使得系统组件里的各元件得到极大的共享。此外,根据系统组件间的数据通信,也可以精确地对结构实现的性能做估计。

2) **对象数量较小**。功能划分的第二个优点是减少了需要划分对象的数量。功能对象的抽象级别比寄存器传输级要高,功能对象的数量和寄存器传输级的结构对象数量相比要少很多。由于对象数量较小,算法的性能得到了提高,同时也更容易引入设计者的交互作用。

3) **软硬件协同设计**。这是功能划分最重要的优点。功能划分处理的对象是功能,允许软硬件划分。映射到处理器上的功能对象可以编译成指令集来实现,而映射到硬件上的对象则可以用结构来实现。很多系统都包含有硬件和软件两部分,这强烈要求划分方法具有在软硬件之间进行划分的能力。

功能划分的传统做法从非正式的自然语言描述开始,这种方法不能满足要求,原因有如下几点:首先,自然语言描述无法被计算机理解,无法实现自动评估和划分,因此划分结果的质量很大程度上依赖于设计者的经验。其次,对自然语言的描述无法在设计早期进行验证,因此系统中的功能错误可能要等到设计的末期才能够发现。再次,自然语言描述通常比较含糊(见 4.5 节),这使整个系统的集成变得困难。当另一个设计者实现系统组件的时候,这种集成问题就会体现出来。不同的设计者对自然语言的描述有不同的理解。在集成系统组件并进行测试时,这种理解的不同就可能表现为设计错误。

为了解决上述局限,人们提出了可执行描述的划分(executable-specification partitioning)的概念。在这种功能划分的特殊方法中,首先用可执行描述语言来说明系统功能,然后从这个描述中获得功能对象,再对其进行划分。和多数基于可执行语言的设计方法一样,这种处理方法也具有其共同的一些优点。首先,这种描述可以被计算机理解,因此可以开发工具来自动完成评估和划分,从而减少了对设计者经验过多的依赖。其次,在设计早期可以通过模拟来对描述进行验证并修正其中的错误,以减少在设计后期进行修正所带来的成本开销。再次,功能对象映射到哪一个系统组件在形式上有详细说明,因此不容易对功能产生不同的解释,能减少集成和实现组件时引发的错误。

对可执行描述的划分问题,已经提出了许多技术。但由于这个领域相对较新,多数技术还集中于一般的功能划分问题的一个较小的子集。在本章后面的几节里,我们将定义功能划分中的关键问题,对基本算法进行概述,然后对近年来所公布的一些划分技术进行介绍。

6.3 划分中的问题

为了便于理解和对比各种不同的划分技术,下面将整个划分问题分为八个基本问题:系统描述抽象级别、粒度、系统组件的分配、度量和评估、目标和接近函数、划分算法、输出,以及控制流程和设计者的参与,如图 6-1 所示。值得注意的是,这些问题并不是作为正式分类。

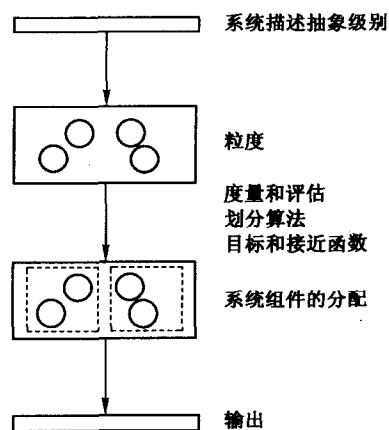


图 6-1 系统划分中的基本问题

175

176

6.3.1 系统描述抽象级别

各种划分技术一般都通过其使用的描述语言来定义划分的输入。然而,同一种语言可以用来描述多个不同的概念模型,这些模型分别需要不同的划分方法,因此单凭语言本身不足以定义输入。比如用 VHDL 来描述傅里叶变换,就可以分别从算法级、寄存器传输级,甚至门级等不同级别进行描述。

因此,根据概念模型的抽象级别来定义输入更为恰当。比较粗略的可以将抽象级别定义为系统描述中低复杂度结构对象数量的度量,抽象级别越低,说明低复杂度结构对象越多。例如一个描述包含了系统中各个门的互联情况,那么它的抽象级别也就必然很低。

177

在不同划分技术中,考虑到了多种抽象级别的描述,由高到低大致有如下几种:

1) **任务数据流图**。输入代表着一个数据流图,其中每个操作代表一个任务(见 2.4.1 节),而任务则是比一个基本算术运算复杂得多的计算。这种模型没有定义每个任务的具体计算过程,但定义了各个任务之间的数据传输情况,以及每个任务的时延、面积等参数。

2) **任务级**。输入表示一系列任务,其中每个任务通过一个顺序程序来描述,这在 2.7.3 节中有详细介绍。任务级通常也叫做行为级。

3) **算法级数据流图**。输入为数据流图或加减法之类的算术运算,还可能带有一些控制操作,详细说明请参见 2.7.1 节。由于多数高层次综合的研究和工具中都使用这种模型,因此它也是划分技术中最通用的一种模型。这种模型也叫做控制/数据流图,即 CDFG。

4) **带数据通道的有限状态机**。输入为有限状态机,可能含有复杂的表达式,这些表达式在状态内或状态转换间进行计算,如 $a = b + c \times d$,对于有限状态机在 2.3.1 节中有详细介绍。显然这里也可以使用传统的有限状态机,它只允许有布尔表达式,在某些情况下也在使用。

5) **寄存器传输级**。输入为一系列的寄存器传输,即对每个机器状态都提供寄存器之间的传输过程。

6) **结构级**。输入为物理组件间的结构互联,也叫做网表。如前所述,这些组件本身可以是不同抽象级别。

显然,一个描述输入中的不同部分可以属于不同的抽象级别,例如一部分输入是任务级的描述,而另一部分则使用设计好的门级互联结构描述。

不同是抽象级别代表了设计中不同的中间实现体。由于设计者最开始从较高抽象级别中得到设计概念,因此通常的设计都从较高的抽象级别开始,在后续较低抽象层次里加入结构细节的信息。这样,划分输入抽象级别的不同也就能代表划分之前已完成设计的多少。

178

6.3.2 粒度

划分中的第二个问题就是把输入的功能对象分解成为多大的粒度。如前文所述,描述首先将被分解为功能对象,然后再在系统组件间进行划分。所谓分解的粒度,就是每个对象中所含描述规模的度量。粗粒度意味着每个对象中含有大量描述,而细粒度则表示对象中仅含有少量描述,从而具有较多的对象。

对于特定抽象级别的输入,只有一种分解粒度较为合理。比如任务级数据流图通常分解为任务级,而算法级数据流图一般则分解为算法操作级。

任务输入抽象级的分解具有较大的灵活性。既可以只考虑任务本身,又可以将任务分解

为多个过程,还可以将任务和过程进一步分解为 if-then-else 或者 loop 循环体之类的语句块。此外,甚至还可以将每个语句单独考虑。

显然,对输入分解的粒度越细,就能获得更多的划分结果,也就能获得更好的优化结果。然而细粒度也有着一些缺点。首先细粒度分解导致划分对象增多,这使得划分算法需要更多的时间进行计算,否则得到的结果会不理想。其次,设计者不容易辨识细粒度对象,因此也就难于支持手动交互。再次,细粒度划分的结果对设计者来说不易理解,对其进行手动设计或修改都较困难。最后,对系统的评估也较困难。

为了理解低粒度下评估的困难性,下面来看划分中这个硬件面积评估的例子。假设有一个精确的评估算法,其计算复杂度为 $O(n^2)$,其中 n 为算法操作的数量。如果在操作级别较低的层次上进行划分,假设共有 10 000 个操作,那么对每次划分结果进行面积评估就需要 100 000 000 次计算。如果需要进行多次划分并进行评估的话,这种计算量显然是不现实的。相反,如果在较高级别上进行划分,就能在划分之前对每个任务使用该算法进行评估。虽然评估的计算复杂度很高,但只允许一次。评估的结果是获得每个任务所需面积的估计值,也许还有更多的面积信息,这将在 7.5.4 节中进行进一步讨论。对于给定任务级划分的面积评估,只需要对其中每个任务的评估进行合并即可。这和对每次划分都在操作级进行评估相比,显然要简单得多了。

179

6.3.3 系统组件的分配

划分技术必须指定功能对象映射到的系统组件类型。一些通用组件包括处理器、ASIC、存储器、寄存器文件、数据通路和总线等。

系统组件的分配就是对一个给定的设计,在允许的系统组件类型里选择一些,并确定各类组件所需的数量。这种选择组件的过程就叫做分配(allocation)。为实现一个给定的描述,有着多种不同的分配方案,它们在成本、性能方面各有不同。此外还需考虑功耗、设计时间和厂商的可靠性等问题。分配通常由手动完成,或者结合划分算法共同完成。

6.3.4 度量和评估

每种划分技术都需要定义一些划分的属性,来评价划分结果的好坏,这些属性就称做度量(metrics)。常用的度量有成本、执行时间、通信比特率、功耗、面积、引脚、可测试性、可靠性、程序大小、数据大小,以及存储器大小等。

一些技术每次只组合一个对象,直到得到完整的划分。由于上面提到的度量都是对完整划分提出的,对于这类划分技术就需要一些新的度量,来预测任意两个对象所做的组合是好是坏,这些度量称做接近度量(closeness metrics)。如一个结构对象和一组特定的结构对象有很多互联线,那么它们的关系就比较密切。如果一个任务和一个任务组共享较多变量,那么它们的关系也比较密切。有关接近度量的详细讨论请参见 6.5.2 节、6.5.3 节和 6.7.3 节。

180

度量的值必须通过某种方法来计算,但这并不是太容易。这是由于所有度量都是根据实现功能对象的结构(或软件)来建立的,而在划分中这些实现并不存在。要计算度量,可以采用下面的两种方法。其一是实际实现所划分的系统,这虽然能得到精确的度量值,但需要花费数天以至数月的时间来实现。如果使用非平凡划分算法,将产生成千上万种划分结果,这些结果都需要进行度量,对这种情况,即使只需要几分钟来建立实现,所需的时间也是无法忍受的。

第二种方案是粗略地快速实现系统,在其基础上计算度量值。粗实现包含了设计中主要

的寄存器传输级元件,但并不涉及其中布线或逻辑优化等耗时的细节。通过这种粗实现来计算度量值就叫做评估(estimation),在下一章里将对评估进行详细介绍。

评估的速度和精确度是互相矛盾的。为了达到较快的速度,必然会忽略粗实现中的一些细节;反之为了达到较高的精度则必须考虑这些细节。对于每种度量,都有不同的实现模型,在不同程度上对速度和精度进行折中。

对于不精确的评估,只要能够对两个不同划分的相对好坏做出正确的判断,在某些情况下也是可以接受的。评估所具有的这种预测两个划分相对好坏的能力,就叫做具有高保真度(fidelity)。

值得注意的是,划分的输入描述抽象程度越低,评估的结果就越精确。这是因为评估会去预测后续设计步骤的结果,而低层描述的后续设计和高层描述相比会少得多,从而能避免过多错误的预测。

181

6.3.5 目标函数和接近函数

划分通常使用多种不同的度量,如费用、性能和功率,这些度量往往是相互矛盾的。为了定义划分结果的好坏,需要把多种度量组合成一个函数,称为目标函数(objective function),其函数值称为成本(cost)。

由于各种度量的重要性不同,如何将多种度量组合成1个值也是一个重要的问题。多数方法使用加权和和目标函数,即用权重来表示各个度量的相对重要性,将度量值和其权重的乘积和作为目标函数的函数值。一个加权和和目标函数的例子如下所示:

$$Objfct = k_1 \cdot area + k_2 \cdot delay + k_3 \cdot power \quad (6-1)$$

这个函数包含有面积(area)、时延(delay)和功率(power)三个度量值,其权重分别为 k_1 、 k_2 和 k_3 ,其和为目标函数值。如果 k_1 比 k_2 和 k_3 大,就意味着面积比其他两个度量更重要。

由于多数设计决策都是受约束的驱动,实际上很少使用公式(6-1)那样的简单方程。在目标函数里必须加入约束,这样得出的划分结果和不加约束的结果相比,能更好地满足约束的要求。如上述目标函数可以扩充为下面的形式:

$$\begin{aligned} Objfct = & k_1 \cdot F(area, area_constr) \\ & + k_2 \cdot F(delay, delay_constr) \\ & + k_3 \cdot F(power, power_constr) \end{aligned} \quad (6-2)$$

式中的函数 F 表明了度量的估计和给定约束之间的关系,通常在不满足约束的时候返回非零值,而在满足约束的时候则返回零。因此,即使两种划分结果的功率不一样,只要它们都满足功率的约束,那么在对比这两种划分的时候功率就不起作用。当各种约束都满足的时候,这种形式的函数 F 使得目标函数返回零,于是划分的目标就是获得成本为零的划分方案。

182

组合各种度量的另一个问题就是各个度量单位的归一化。比如面积的约束是9000个面积单位,评估的结果可能是10000个面积单位;而时延的约束是1个时间单位,评估的结果为10个时间单位。假设目标函数中的各个系数 k 都相同,那么应该更看重减小时延而不是面积,因为时延的评估结果是其约束的10倍。特别地,时延减少5个时间单位比面积减小5个单位要好得多,因为前者减小了50%的时延,而后者仅仅减少了1%的面积。如果将每个函数 F 的值除以其度量约束值,就能够很清楚地看出前者的优势所在了。

一方面目标函数将各个度量结合在一起,来评价划分的结果;而另一方面,接近函数

(closeness function)将关系紧密的度量结合在一起,在完成整个划分以前来衡量是否需要将对象合为一组。权重和目标函数的归一化这些讨论同样也适用于接近函数。

6.3.6 划分算法

对于给定的一组功能对象和系统组件,划分算法将搜索最好的划分方案,最好意味着该方案的目标函数具有最小的成本。

显然,通过穷举法产生所有可能的划分方案,然后对每个方案的目标函数值进行计算,就可以得到最优解。然而这种方法计算量太大,没有实际意义。假设有 n 个对象和 m 个组件,就有 m^n 种可能的映射。考虑一个较小的规模,当 $n=20$ 、 $m=4$ 时,有一万多种映射结果。这决定了划分算法的核心,即如何在所有可能划分情况中选取一个子集来进行检测。

划分算法通常可以分为两类,即构造性算法和迭代算法。构造性算法将全部对象分组来获得整个划分,并使用接近度量来指导对象的分组,从而获得较好的划分结果。构造性算法的计算时间主要用于构造这些数量较少的划分。

迭代算法需要对已有的划分结果进行修改,以期得到改善,并通过目标函数来对划分进行评价,这样得到的结果比构造性算法中的接近函数要精确得多。迭代算法的计算时间主要用于评价大量的划分方案。迭代算法和构造性算法相比,主要区别在于如何修改已有的划分,以及如何接受或拒绝较坏的修改。为了尽可能减少计算量,在搜索过程中需要逃逸出局部最小点。图 6-2 给出了一系列解的移动和每次移动后的成本值,由此来说明局部最小点的概念。对划分来说,移动就是对已有结果进行修改,并将重新对象映射到不同的组件上。在图 6-2 中,最左边一个点代表初始划分的成本,前两次移动都使得成本有所降低,达到点 A。接着的两次移动虽然增大了成本,但其后续将移至成本更小的点 B。这里点 A 就是一个局部最小(local minimum)点,而点 B 则是全局最小点。一类迭代算法在搜索中只接受比当前成本更低的解,这类算法称为贪心算法(greedy algorithm),这类算法无法逃出局部最小点;而另一类算法则能逃出局部最小,通常称为爬山算法(hill-climbing algorithm)。

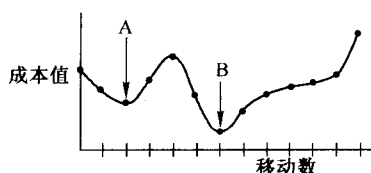


图 6-2 迭代划分中逃出局部最小点

在实际使用中,经常将构造性算法和迭代算法结合起来。我们将在 6.4 节里对基本划分算法进行进一步阐述。

6.3.7 输出

每种划分技术都必须定义其输出的表示格式和可能应用。格式可以是一个列表,指示每个功能对象映射到了哪个系统组件上;也可以是另一种新的输入描述,含有系统组件的结构对象,并通过功能对象的映射关系来定义组件的功能。输出的可能应用描述了输出信息在后续设计过程中所扮演的角色。例如输出可以作为功能描述提供给设计者,来完成每个组件的具体实现,也可以作为综合工具的输入,甚至给综合工具提供启发式信息,来帮助综合工具选取信息中的有用部分。

6.3.8 控制流程和设计者的参与

通过上述讨论可以看到,对于一个给定的描述,要对其中的系统组件进行划分,需要做出很多的决定,包括选择对象的粒度、分配系统组件、选择质量度量、选择目标函数或接近函数以

及选择划分算法等。这些决定可以根据上述顺序或者其他顺序来执行,其中某些决定需要多次执行。比如可以用下面的顺序来完成整个流程:选择粒度、选择接近度量、选择接近函数、分配组件、执行构造性划分算法、重新选择其他的接近函数、重新执行构造性划分算法、选择目标函数、执行迭代划分算法、重新分配系统组件、重新执行迭代划分算法。整个过程的控制流程可以不同。不同的顺序会导致不同的结果,因此划分技术必须指明这些决定的顺序,以此在特定设计目标(如性能最优)下得到一个较好的结果。

在实际应用中,划分系统必须允许设计者的交互。交互可以分为两类。一类是指示,描述了设计者可以手动执行的动作,如分配、将指定对象移动到指定组件上、将评估替换为更好的评估等。第二类交互是反馈,描述了当前设计提供给设计者的信息。比如一个图可以表示对象间的连线数目,直方图可以表示设计约束的不满足程度。

185

6.3.9 典型系统配置

图 6-3 中描述了一个可执行描述的划分系统的典型配置。输入(input)被转换为划分的内部模型(model),二者在功能上完全等价,该模型将用于之后的划分算法(algorithm)。划分算法需要评估器(estimator)和目标函数(objective function),其中评估器对模型进行计算,并将结果传递给目标函数;目标函数用于评估。划分模型会被转换为输出(output),以适应后续的设计和分析。后续实现中获得的各种度量也叫做设计反馈(design feedback),可以在后续划分中用以提高设计质量。设计者可以通过用户接口(user interface)来和系统中的不同部分进行交互。

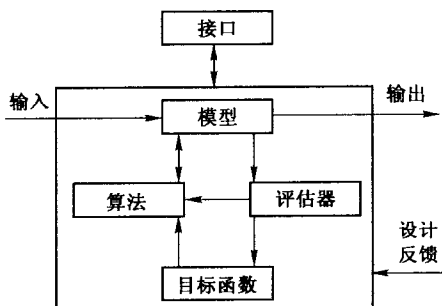


图 6-3 划分系统的典型配置

6.4 基本划分算法

划分算法将每一个功能对象映射到一个组上,每组代表一个系统组件。从理论上来说,算法产生的划分结果能使目标函数获得最小成本。形式化的划分问题定义如下所示:

186

定义 6.4.1: 划分问题

给定一个目标集合 $O = \{o_1, o_2, \dots, o_n\}$, 求划分 $P = \{p_1, p_2, \dots, p_m\}$, 使其满足 $p_1 \cup p_2 \cup \dots \cup p_m = O, \forall i \neq j, p_i \cap p_j = \emptyset$, 并使目标函数 $\text{Objfct}(P)$ 成本最小。

接下来将讨论随机映射、层次化结群、多级结群、成组移动、比率切割、模拟退火、遗传进化和整数线性规划等多种划分算法。

6.4.1 随机映射

一种常用的构造性算法是随机映射,它把每个对象随机地和给定的组件进行搭配。这种算法通常用来获得迭代算法的初始划分。其计算复杂度为 $O(n)$, 其中 n 为对象数量。

6.4.2 层次化结群

层次化结群是一类常用的构造性划分算法[Joh67, LT91, MK90, CB87, GDWL91], 它使用接近度量来将对象进行分组, 因为其他度量无法在划分完成以前得到其值。引入接近度

量是为了获得一个好的全局度量值。这种方法将关系密切的对象进行组合,然后重新计算组合后的接近性,并重复该操作直到满足终止条件为止。

算法 6.4.1 使用 6.4.1 中定义的标记,详细描述了一个层次化结群算法。算法中的过程 ComputeCloseness 用来计算两个对象 o_i 和 o_j 之间的接近性,计算结果保存在集合 C 的元素 $c_{i,j}$ 中。对初始对象,ComputeCloseness 根据 6.3.5 节中描述的接近函数来确定其接近性。然而对于组合后的层次化对象,有多种方法来计算其接近性。一种方法是使用接近函数,但是反复调用该函数需要大量的计算;另一种方法是估计层次化对象 o_{ij} 和 o_k 间的接近性,并将其作为 $c_{i,k}$ 和 $c_{j,k}$ 之间的最小、最大、平均或是总接近性。

187

过程 FindClosestObjects 搜索给定集合中接近性最高的一对对象;过程 Terminate 在算法结束的时候返回真值。Terminate 的一种普遍形式是在所有对象都组合到了一定数量的组后返回真,另一种形式是当所有接近性的值都低于某一数值时返回真,这个特定的数值叫做接近性阈值(closeness threshold)。

188

算法 6.4.1: 层次化结群

```

/* 初始化,将每个对象作为一组 */
for 每一个  $o_i$  loop
     $p_i = o_i$ 
     $P = P \cup p_i$ 
end loop

/* 计算对象间的接近度 */
for 每一个  $p_i$  loop
    for 每一个  $p_j$  loop
         $c_{i,j} = \text{ComputeCloseness}(p_i, p_j)$ 
         $C = C \cup c_{i,j}$ 
    end loop
end loop

/* 合并关系密切的对象,并重新计算接近度 */
while not Terminate(P) loop
     $p_i, p_j = \text{FindClosestObjects}(P, C)$ 
     $P = P - p_i - p_j \cup p_{ij}$ 
    for 每一个  $p_k$  loop
         $c_{ij,k} = \text{ComputeCloseness}(p_{ij}, p_k)$ 
    end loop
end loop
return P

```

算法的开始是初始化,将每一个对象都作为一组,然后计算每一对对象间的接近性。算法的核心部分是将接近对象合并成一个新的对象(实际上是一组),然后重新计算这个新对象和其他每一个对象间的接近性。这个核心部分将重复执行,直到满足终止条件为止。算法的复杂度主要由计算对象对间的接近性决定,为 $O(n^2)$ 。

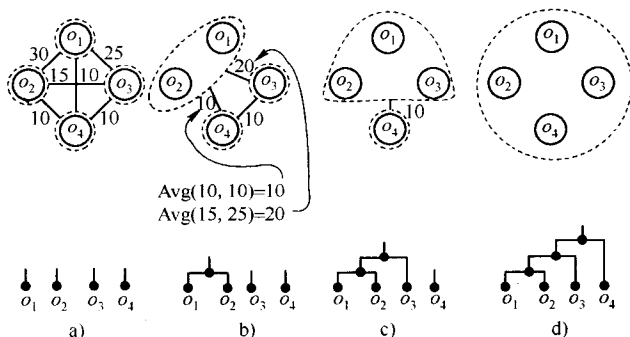


图 6-4 层次化结群

下面来看图 6-4 中的例子,该例含有 4 个对象 o_1 、 o_2 、 o_3 和 o_4 ,它们两两间的接近性值均标在图中。从图 6-4a 中可以看到对象 o_1 和 o_2 关系最密切,其接近值为 30。于是将 o_1 和 o_2 合并为一个新的对象,并计算合并之前这两个对象和图中其他对象接近值的平均值,作为新对象与其他对象的接近值的估计值,得到图 6-4b。接着用同样的处理方法进行合并、计算,可以得到图 6-4c。假设算法终止的条件是任意两个对象间的接近性都不超过阈值 15,那么得到的最终划分结果是 o_1 、 o_2 和 o_3 属于一组,而 o_4 属于另一组,如图 6-4c 所示。

为了保留对象合并的顺序,我们可以对算法进行修改来生成结群树(cluster tree),再进行结群操作,直到系统中只剩一组为止。通过合并的记录可以产生多种可能的划分结果。还是以图 6-4 为例,图 6-4a 中的所有对象 o_1 、 o_2 、 o_3 和 o_4 都作为结群树的叶子结点。在图 6-4b 中,当 o_1 和 o_2 合并成新的对象时,相应地在树中为这两个结点增加一个父结点,来代表合并后的新对象。在图 6-4c 中,上步生成的新对象和 o_3 进行合并,于是在树中为其再增加一个父结点。当合并最后一个对象 o_4 时,增加的父结点就是整个结群树的根结点。对上述例子,整个结群树如图 6-4d 所示。如果在结群树中做一条水平线,对树进行切割,就可以得到一个划分。比如在第二级结点上做一条水平线,如图 6-5a 所示,就可以得到一个划分,将 o_1 和 o_2 分为一组, o_3 和 o_4 各自为一组。这样的线就叫做切割线(cutline),它可以位于结群树中的任何级别,并可以倾斜。通过在结群树中做切割线,可以得到大量的可能划分结果,每个划分结果都包含有若干个组,而每组都由一些关系密切的对象组成。

6.4.3 多级结群

我们可以看到,在层次化结群里,多个接近度量会被合并成一个单一接近值。当有多个接近度量时,一种较好的做法是由一个度量开始进行层次化结群,然后再考虑其他度量。每次根据一个度量进行的结群过程称为一级。可以使用任意数量的级数。这种处理方法被称为多级结群(multi-stage clustering),详细介绍请参见[LT91]。

图 6-5 中描述了一个两级结群。图 6-5a 中是上一节例子中的结群树,切割线表明了系统中共有三个组,这些组在后续的结群中都被看做新的对象,如图 6-5b 所示。在新的一级中,我们使用和上一级不同的度量来重新计算对象间的接近值,并重复此操作,从而得到新的结群树,如图 6-5c 所示。

每一级都含有一个新的层次化结群过程,其复杂度为 $O(n^2)$ 。由于级数为一个较小的数,因此将层次化结群的复杂度乘以一个较小的常数,就可以得到多级结群的理论复杂度,为

$O(n^2)$ 。

结群和随机映射一样,其迭代改进算法都需要一个初始划分,因此必须选择一个构造性算法。如果迭代算法效果很好,最后结果和构造性算法的选择关系就不大。在这种情况下,建议使用随机映射,因为它和结群相比计算量要少。某些实验结果表明,结群和随机映射相比会导致更坏的结果,这是因为结群所得的划分是一个局部最小点,而迭代算法无法逃出这个局部最小。然而另外一些实验结果表明,在对象数目较大的时候,结群能从很大程度上改善最终结果。

190

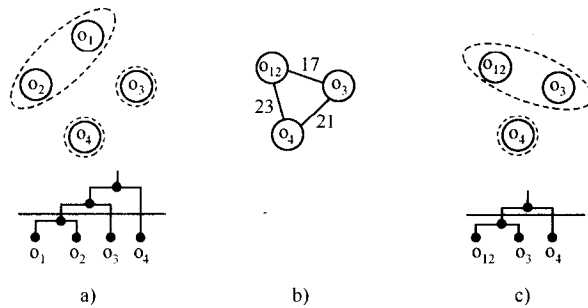


图 6-5 多级层次化结群

6.4.4 成组移动

这种迭代改进算法通常也叫做最小割或者 Kernighan/Lin 算法,是为了改善二路划分的结果而提出的[KL70],而且经过修改来减少其计算量[FM82]并获得更好的结果[Kri84]。该算法的原形是最小化组之间的连线数量,即割值,因此被称为“最小割”算法。后来,该算法引入了割以外的其他多种度量。该算法的控制策略已经对结构划分得到了很好的结果,并且能同样在功能划分中取得成功。

下面将通过对简单二路划分算法进行扩展,来说明该算法的控制策略。二路划分重复执行如下步骤:对每个对象,计算将其移动到另一组而引起的成本减少量,然后选取减少量最多的对象,将其移动到另一组。在某些情况下,移动一个对象并不能减少成本,但后续的两步或者多步移动后可能会使成本减少。该算法无法对这类情况进行处理,也就是说该算法无法逃出局部最小点。

191

成组移动对上述算法进行扩展,在移动的时候选择成本减少量最大或者是成本增加量最少的对象,这样就可以逃出局部最小。为了避免某个对象在两组之间反复移动造成算法中出现无限循环,每个对象都只能移动一次。在每个对象都移动一次后,就可以在计算中所得到的所有划分中选取成本最低的作为最终结果。

整个算法使用这个新的划分作为初始划分进行迭代,直到无法得到成本更低的划分为止。根据实验结果,该过程重复次数一般不超过 5 次[KL70]。

算法 6.4.2 对改善二路划分的成组移动做了详细描述。其中算法的输入为 $P_{in} = \{p_1, p_2\}$, 函数 $Move(P, o_i)$ 将对象 o_i 移动到另一组,并返回新的划分。每个对象 o_i 都有一个标志 $moved$, 用来标明该对象是否移动过,每个对象只能移动一次。变量 $prev_P$ 和 $prev_cost$ 分别代表在迭代的每一步中,移动之前的划分和成本。变量 $bestmove_obj$ 代表移动后能带来最大成本降低的对象,相应的成本用 $bestmove_cost$ 来表示。变量 $bestpart_P$ 代表计算过程

中遇到的成本最低的划分, $bestpart_cost$ 为该划分的成本。

当产生的移动无法改善成本的时候, 算法的最外层循环将停止。在每次迭代中, 都要建立一个 n 步的移动序列, 其中 n 是对象数目。对序列中的每一步移动, 都要试着将每个对象移到另一组, 选出能获得最好结果的一个, 并对其进行标记以免再次移动。如果移动后的成本比以往的都好, 就需要记录这个结果和成本。在得到所有的 n 步移动后, 对保存的最优划分和迭代初始划分进行对比。如果保存的划分较优, 那么就进行下一轮迭代; 否则仍使用上次的划分结果。

算法 6.4.2: 成组移动

```

P = P_in
loop
    /* 初始化 */
    prev_P = P
    prev_cost = Objfct(P)
    bestpart_cost = ∞
    for 每一个  $o_i$  loop
         $o_i.moved = false$ 
    end loop

    /* 建立一个  $n$  步移动的序列 */
    for  $i$  从 1 到  $n$  loop
        bestmove_cost = ∞
        for 每一个没有移动过的  $o_i$  loop
            cost = Objfct(Move(P,  $o_i$ ))
            if cost < bestmove_cost then
                bestmove_cost = cost
                bestmove_obj =  $o_i$ 
            end if
        end loop
        P = Move(P, bestmove_obj)
        bestmove_obj.moved = true
        /* 保存序列中的最优划分结果 */
        if bestmove_cost < bestpart_cost then
            bestpart_P = P
            bestpart_cost = bestmove_cost
        end if
    end loop

    /* 如果找到更优的解则更新 P, 否则退出 */
    if bestpart_cost < prev_cost then
        P = bestpart_P
    else return prev_P
    end if
end loop

```

该算法的复杂度由建立 n 步移动所决定。对每一次移动都需要选择一个最优对象,这平均需要对 $n/2$ 个对象进行检验。假设目标函数需要的计算量的数量级也是 n ,那么算法的复杂度就为 $O(n \times n/2 \times n)$,即 $O(n^3)$ 。7.5.4 节将详细讨论一种增量式评估技术,将目标函数的复杂度降为常数。由于成组移动每次只移动一个对象,如果采用增量式评估技术(见 7.5.4 节),就可以使得目标函数的计算复杂度降低为常数,因而整个算法的复杂度可以降为 $O(n^2)$ 。文献[FM82]针对结构划分对该算法进行了优化,使其复杂度进一步降为 $O(n)$ 。

上述算法很容易扩展为多路划分。在二路划分中,为了得到最优对象,需要试着将每个对象都移动到另一组,然后选出其成本最低的对象;而在多路划分中,需要将每个对象都移动到其余的每一组。于是需要引入一个新的变量 *bestmove_group*,来指明最好的移动方案中对象会移动到那一组里。算法的复杂度将乘以组数。假设对于二路划分的算法复杂度为 $O(n^2)$,那么对多路划分的算法复杂度将变为 $O(mn^2)$,其中 m 为划分的组数。

在多路划分的另一种扩展方案中,先根据二路划分的算法生成两组,然后再使用二路划分算法,将其中每组都继续划分成两组。如此重复,直到划分的组数满足要求为止。由于划分的组数 m 一定,因此该算法的复杂度也为 $O(mn^2)$ 。

6.4.5 比率切割

比率切割也是一种构造性算法,最初是针对结构划分提出的,对于对象数量很多的情况有很好的效果。下面介绍该算法的一般形式,它也适用于功能划分。该算法将对象进行分组,直到满足终止条件为止,也就是说没有对象被认为和其他对象关系足够密切的时候才结束。该终止条件是形成自然组,而不像 6.4.2 节中的层次化结群算法,需要指明组的数量或者是接近性阈值。

这种新的终止条件依赖于新的划分度量的定义,这种度量可以从切割数扩展而得,切割数就是穿越一组边界的所有边权重的总和。可以通过组的切割数来评估划分,切割数越小,就意味着联系密切的对象已经位于同一组里。然而,如果只有切割数这一个度量,那么所有对象最终合为一组,而不管对象间的关系是否密切。为了避免这种情况,需要对一组里含有对象数目,即组的规模做一定的限制,但这种限制可能导致无法获得组大小不平衡的最优划分。基于以上考虑,比率切割的目标就是将对象进行组合来减小切割数,但不合并关系不密切的对象,也没有组大小的限制。

为了达到上述目标,这里用一个新的度量比率(ratio)来代替切割数。对划分 $P = \{p_1, p_2\}$,令 $cut(P)$ 为连接 p_1 和 p_2 的边的权重总和, $size(p_1)$ 和 $size(p_2)$ 分别表示 p_1 和 p_2 的大小。划分的比率定义如下,该值越小越好:

$$ratio = \frac{cut(P)}{size(p_1) \times size(p_2)} \quad (6-3)$$

在结群中,需要组合对象以减小切割数,而又不合并关系不密切的对象,比率在相互矛盾的两方面间能取得平衡。其中分子倾向于组合对象,组间的边权重和越小, $cut(P)$ 也就越小;分母则用来维持多组大小的平衡,组大小越平衡, $size(p_1) \times size(p_2)$ 就越大。

有了比率这个新的度量,就可以寻找具有最小度量的划分,这种划分称为比率切割。[KC91]中提出了一种启发式的算法,它将比率直接作为目标函数,首先获得初始划分,然后多

195 次运用成组移动。

6.4.6 模拟退火

在 6.4.4 节中我们可以看到,如果接受成本增大的移动方案,成组移动就可以逃出局部最小,从而获得成本更低的划分。并且在整个序列中每个对象只移动一次,降低了算法的计算复杂度。模拟退火算法[KG83]同样接受成本增加的移动,但和成组移动不同的是,模拟退火中每个对象可以移动多次,随着时间的推移,接受成本增加的移动的概率逐渐减小,以此降低算法的复杂度。算法模拟物理中的退火过程,就是将材料熔化,再以足够慢的速度降低温度,这样在每个温度下都能够达到平衡状态,通过这种方法来使材料达到能量最低的状态。

模拟退火算法从一个初始划分和初始模拟温度开始,逐渐降低温度。在每个温度下随机产生移动。任何移动都可能被算法所接受,即使该移动会导致成本上升,但接受成本上升移动的可能性随着温度的降低而减小。

算法 6.4.3 详细描述了一个模拟退火算法。变量 *temp* 记录当前的模拟温度;函数 *RandomMove* 在当前划分 *P* 中随机选择一个对象,将其移动到另一组,产生一个新的划分 *P_tentative*;变量 *cost* 和 *cost_tentative* 分别为当前成本和试探划分的成本;变量 $\Delta cost$ 用来保存试探划分和当前划分的成本差,可能为负值;函数 *Accept* 根据成本的增加值和当前温度,由下式[KG83]来判断是否接受该移动:

$$Accept(\Delta cost, temp) = \min(1, e^{-\frac{\Delta cost}{temp}}) \quad (6-4)$$

当 $\Delta cost$ 为负时,意味着试探划分比当前划分更好, *Accept* 将返回 1;否则将返回一个位于 $[0, 1]$ 间的值;函数 *Random*(0, 1) 返回一个 $[0, 1]$ 间的随机数;函数 *Equilibrium* 判断在当前温度下,划分过程是否达到平衡,如果多次迭代都不能改善成本,就可以认为已经达到平衡了;达到平衡后,通过函数 *DecreaseTemp* 来降低温度,通常根据 $temp_new = \alpha \times temp_old$ 来确定新的温度,其中 $0 < \alpha < 1$;函数 *Frozen* 确定是否达到最低温度值,也就是判断算法是否结束,其中最低温度通常为 0。

196

算法 6.4.3: 模拟退火

```
temp = 初始温度
cost = Objfct(P)
while 没有达到最低温度 loop
    while 没有平衡 loop
        P_tentative = RandomMove(P)
        cost_tentative = Objfct(P_tentative)
        Δcost = cost_tentative - cost
        if (Accept(Δcost, temp) > Random(0, 1)) then
            P = P_tentative
            cost = cost_tentative
        end if
    end loop
    temp = DecreaseTemp(temp)
end loop
```

在某一特定温度下,都产生随机移动并选择性进行接受,以此来试着改善划分,直到达到

平衡。当 *Accept* 的返回值大于 *Random*(0, 1) 的返回值时, 接受该移动。若移动对划分有改善, 那么 *Accept* 将返回 1, 这种移动一定能被接受。达到平衡后将降低温度, 继续执行上述改善步骤, 直到达到最低温度。

理论研究[RSV85, Len90]表明, 如果在每一个温度下都达到了平衡状态, 并且温度降低速度无限低, 模拟退火算法能够逃出局部最小点, 并最终到达全局最小点。但这需要无限多的温度, 在每个温度下也需要无限次迭代, 显然是不现实的。于是人们提出了许多启发式算法 [OvG84, HRSV86], 来控制模拟退火过程。这些启发思想定义了平衡状态以及如何降低温度。算法复杂度依赖于 *Accept*、*Equilibrium*、*DecreaseTemp* 和 *Frozen* 这些函数的具体形式, 尽管这些函数通常都选用多项式复杂度的算法 [Len90], 但整个算法的复杂度仍可以是指数级到常数级间的多种形式。一般而言, 模拟退火能获得较好的结果, 但需要较长的计算时间。

197

6.4.7 遗传进化

成组移动和模拟退火算法都通过移动一定数量的对象来改善当前划分, 保存计算过程中的最好划分, 并以这个最好划分为基础继续迭代。然而并不需要计算过程中保存的划分数目限制为一个。

有一类算法根据遗传进化过程, 保存迭代过程中的一系列划分结果, 算法中的这一系列划分也称为一代。遗传算法在某一代的基础上, 通过模拟自然界的三种进化方式来产生下一代。第一种方法是选择(selection), 即随机选择一个低成本的划分 P 作为下一代的解, 也就是将这一代中的较强者保存在下一代中。第二种方法是杂交(crossover), 即随机选择两个较好的划分 P_a 和 P_b , 将某一划分的一些特性复制到另一划分中, 如将 P_a 中的组 p_i 复制到 P_b 中, 也就是将这一代中较强者将通过混合后保存在下一代中。第三种方法是变异(mutation), 即随机选择一个划分, 并随机移动组间的某些对象来进行修改。

算法 6.4.4 详细描述了一个遗传划分算法, 它是对 [KV93] 中的算法经过改变而得。函数 *CreateRandomPart*(O) 返回给定对象集 O 的一个随机划分; 函数 *Select*(G , num_sel) 在 G 当代的基础上, 通过选择的方法产生 num_sel 个划分; 函数 *Cross*(G , num_cross) 在 G 当代的基础上, 通过杂交的方法产生 num_cross 个划分; 函数 *Mutate*(G , num_mutate) 在 G 当代的基础上, 通过变异的方法产生 num_mutate 个划分; num_sel 、 num_cross 和 num_mutate 都是算法的输入; 函数 *BestPart*(G) 返回 G 中成本最低的划分; 函数 *Terminate* 在满足终止条件, 即算法结束的时候返回真值, 一种常用的终止条件是最优划分存活了一定代数而没有被淘汰; 变量 P_best 保存所遇到的成本最低的划分; 输入 gen_size 指明了第一代里出现的划分个数。

198

算法 6.4.4: 遗传进化

```
/* 产生第一代, 具有 gen_size 个随机划分 */
G =  $\Phi$ 
for i 从 1 到 gen_size loop
    G = G  $\cup$  CreateRandomPart(O)
end loop
P_best = BestPart(G)

/* 世代进化 */
```

```

while 没有终止 loop
    G = Select(G, num_sel) ∪ Cross(G, num_cross)
    Mutate(G, num_mutate)
    if Objfct(BestPart(G)) < Objfct(P_best) then
        P_best = BestPart(G)
    end if
end loop

/* 返回最后一代的最优划分 */
return P_best

```

算法首先从第一代里随机生成 gen_size 个划分,然后通过选择、杂交和变异来生成新一代,并重复此步骤,直到满足终止条件为止。最后算法返回计算过程中所遇到的最优划分。

199 遗传算法的复杂度和函数 *Terminate* 的形式有很大关系。和模拟退火类似,遗传算法通常也能得到较好的结果,但需要较长的计算时间。另外,由于遗传算法需要保存多个划分结果,因此需要较多的内存。

6.4.8 整数线性规划

划分问题也可以通过线性规划进行求解。一个线性规划有一系列变量,一系列线性不等式作为变量值的约束,以及一个含有这些变量的简单线性方程作为目标函数。线性规划的目标是寻找满足所有约束的变量,使目标函数最小化。

线性规划可以如下进行描述。求变量 v_1, v_2, \dots, v_n 的非负值,使目标函数 $\sum_{j=1}^n k_j v_j$ 最小,其中 k_j 均为常数。变量满足 m 个形如 $\sum_{j=1}^n a_{ij} v_j \leq b_i$ 的不等式,其中 a_{ij} 和 b_i 都是常数,且 $1 \leq i \leq m$ 。其中目标函数和约束不等式的左端都是变量的简单线性函数,函数中的变量都和一个常数相乘。

对于划分问题的线性规划形式,使用变量来表示划分或度量的评估值。例如划分中将对象 o_i 映射到组 p_j 上,就可以引入一个整数变量 $map_i = j$,这样每个对象都对应一个 map 变量,这些变量共同表示了整个划分结果。此外还可以定义变量 $area_j$,用来表示组 p_j 的面积评估值。使用线性不等式来表示约束,如组 p_j 的最大面积为 A ,对应的约束不等式为 $area_j < A$ 。目标函数可以用 6.3.5 节中的形式,但目标是满足所有约束。

200 如果在线性规划中的变量只能取整数值,那么就叫做整数线性规划,即 ILP(integer linear program)。划分问题可以转化为一个 ILP,通过不同的方式来求解,如分支定界、拉格朗日松弛法等。这些内容已超出了本书的范围,这里不再赘述。由于 ILP 问题是 NP-hard 问题 [GJ79],通常采用启发式算法进行求解。与模拟退火类似,线性规划可以有不同的复杂度 [Len90]。尽管通过 ILP 的形式可以获得划分的较好结果,但和上文提到的构造性和迭代划分算法相比,求解 ILP 需要较多计算时间。

6.5 硬件功能划分

接下来,我们将对一些已发布的硬件组件的系统功能划分算法进行简短描述,包括其研究动机,以及 6.3 节中提到的相关问题。

6.5.1 Yorktown 硅编译器

1. 研究动机

Yorktown 硅编译器也称为 YSC[CvE87, CB87], 它从一个功能描述产生其逻辑级表达式, 然后使用逻辑综合来优化该表达式, 再根据工艺将逻辑运算映射到门上。操作的数量通常很大, 因此需要很长的运算时间和大存储空间。为了缓解这个问题, 可以将逻辑算符进行划分, 然后分别对每组划分结果进行逻辑综合。

2. 概述

如图 6-6a 中例子所示, YSC 接收功能描述, 然后将其转换为一系列表达式, 这些表达式包含加、减和移位等算术操作, 以及与、或等布尔操作。划分的输入就是这些表达式。图 6-6b 中以数据流图的形式给出了一个例子。

划分的对象是给定一系列表达式中的操作, 它们将映射到模块上, 这些模块代表着组合逻辑块, 然后对这些逻辑块分别进行逻辑综合。模块的数量由划分算法决定。

YSC 使用层次化结群算法, 并用接近阈值作为终止条件, 如 6.4.2 节所示。算法中用到的接近函数组合了多个度量, 包括两个操作之间的互联线数、两个操作与其他操作间共有的互联线数、两个操作的逻辑共享性, 以及一个操作群里的晶体管数目等。接近函数试着合并逻辑联系紧密的块, 并保持划分面积的平衡性, 该函数如下所示:

$$Closeness(p_i, p_j) = \left(\frac{Conn_{i,j}}{MaxConn(P)} \right)^{k_2} \times \left[\frac{size_max}{Min(size_i, size_j)} \right]^{k_3} \times \left[\frac{size_max}{size_i + size_j} \right] \quad (6-5)$$

其中各表达式意义如下所示:

- $Conn_{i,j}$ $= k_1 \times inputs_{i,j} + wires_{i,j}$,
- $inputs_{i,j}$ 等于组 p_i 和 p_j 共享的公共输入数量,
- $wires_{i,j}$ 等于组 p_i 和 p_j 之间的连线数量,
- $MaxConn(P)$ 等于划分 P 中, 所有组对 p_x, p_y 间的最大连接数 $Conn$,
- $size_i$ 等于组 p_i 的面积估计值, 以晶体管数来表示,
- $size_max$ 等于允许的最大组面积,
- k_1, k_2, k_3 常量。

该式的第一个部分支持合并共享公共数据的群, 即互联性较高的群。第二部分支持合并较小的群, 这可以使最后的结果较大, 从而得到平衡的划分。第三部分避免划分中的群面积超过给定的限制过大。

下面来看图 6-6 中的例子。首先计算 b 中各个操作间的接近值。从图中可以看到, “-”和“<”间有四位的连线, 因此 $wires_{-, <} = 4$; 而其他操作间均无连线, 因此其余的 $wires$ 值均为 0。另外, “+”和“=”间的共同输入 $inputs_{+, =} = 4 + 4 = 8$, 其余操作对的共同输入均为 0。由上可得 $MaxConnectivity(P) = 8$ 。假设操作“+”、“=”、“-”和“<”分别需要 120、140、160 和 180 个晶体管, 那么可以获得下面的接近值:

$$Closeness(+, =) = \frac{8+0}{8} \times \frac{300}{120} \times \frac{300}{120+140} = 2.9$$

$$Closeness(-, <) = \frac{0+4}{8} \times \frac{300}{160} \times \frac{300}{160+180} = 0.8$$

其他操作对间的接近值都为0。各操作对间的接近值如图 6-6c 所示。

图 6-6d 中显示了层次化结群的划分结果,其接近阈值为 0.5。操作“+”和“=”属于同一群,“-”和“<”属于另一群。

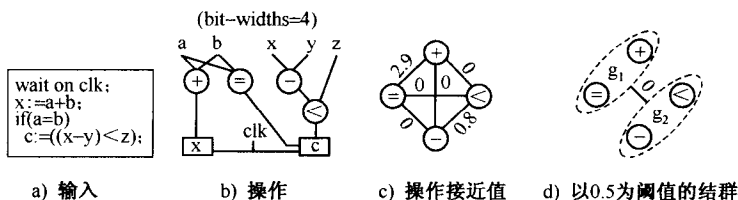


图 6-6 YSC 划分例子

如果考虑到逻辑优化,那么划分结果还可以得到进一步的改善。某些操作能共享逻辑,因此更需要进行逻辑优化。这些操作应该具有较大的接近值,使其更容易得到合并。例如相等和加法操作可以共享逻辑,相等操作可以通过按位求异或,然后将所有输出的反码求“与”从而得到结果,而加法操作可以通过异或产生每一位对的和。如果两个操作具有相同的输入,那么逻辑优化就可以让这两个操作共享异或门,从而减少需要的逻辑数。

为了将这种逻辑优化可能性结合到划分中,需要定义一个新的接近度量,即相似度(similarity),两个原子操作间的相似度表明了这对操作可被逻辑优化的可能性。在划分之前,对每一对操作进行逻辑综合,来获得其相似度。如果综合结果表明它们可以共享逻辑,那么这对操作就具有较高的相似度。在划分的时候,操作对间的接近值与其相似度相乘,使相似的操作获得更高的接近值,如下所示:

$$Closeness(p_i, p_j)' = similarity_{i,j} \times Closeness(p_i, p_j) \quad (6-6)$$

图 6-7 中给出了一个使用相似度划分的例子。图 6-7a 为图 6-6 中的例子,并将阈值由 0.5 改为 3.0,因此各个操作都不需要进行合并。然而,“+”和“=”可以进行合并,因为它们可以在共享逻辑中获得很好的逻辑优化结果。在引入了相似度的理论后,图 6-6b 中给出了各个操作间的相似度值,其中略去了相似度的计算过程。在图 6-6c 中,每个接近值都乘以了相应的相似度,这使得“+”和“=”之间的接近值超过了阈值 3.0。在使用了相似度后,新的结群结果如图 6-6d 中所示,可以看到“+”和“=”位于同一组内,这和我们期望的结果相一致。

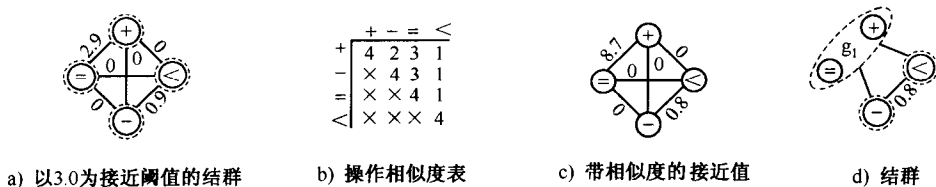


图 6-7 带相似度的 YSC 划分

由于这种划分技术中对象的抽象级别很低,因此手动划分不太现实。设计者的交互作用仅限于选择接近函数中的常数值和最大尺寸约束等。

6.5.2 BUD

1. 研究动机

在物理设计中,布局和布线等对面积和时延有着很大的影响,BUD[MK90, McF86]就是为了解决该问题而提出的。系统设计者在做高层决定的时候,使用到很多“自顶向下”的信息,这些信息也同样在综合工具中广泛使用。因此在高层次综合里必须对面积和时延做出精确的评估。

在高层次综合中的每一步,都需要大量的计算量才能做出评估,这需要研究新技术来减少评估时间。BUD通过对CDFG操作生成少量划分,然后对每个划分分别进行评估,从而减少了评估时间。对一个给定的划分应用简单调度和分配算法,使该划分符合调度和分配。由调度和分配的结果,就能够对面积和时延做出估计。

2. 概述

BUD接受简单的输入任务,将其转换为CDFG,然后将CDFG分解为操作,再进行划分。操作将被划分到数据通路模块上,且模块数量预先不确定。划分结果定义了分配和绑定,对高层次综合中的调度有很大的影响。

BUD使用层次化结群算法来对操作进行划分,并使用两个操作间的接近度量,包括连线数量、共有连线数量、并行执行的可能性,以及功能单元的面积和时延等。使用到的接近函数如下所示:

$$\begin{aligned} Closeness(o_i, o_j) = & \left(\frac{FU_cost(o_i) + FU_cost(o_j) - FU_cost(o_i, o_j)}{FU_cost(o_i, o_j)} \right) \\ & + \left(\frac{Conn(o_i, o_j)}{Total_conn(o_i, o_j)} \right) \\ & - N \times (Par(o_i, o_j)) \end{aligned} \quad (6-7)$$

其中各项意义如下所示:

o_i 第 i 个操作,

$FU_cost(o_i, o_j)$ 以时延和面积为单位的成本,完成给定操作所需功能单元的最小数,

$Conn(o_i, o_j)$ o_i 和 o_j 共享的连线数量,

$Total_conn(o_i, o_j)$ 连接 o_i 和 o_j 的连线总数,

$Par(o_i, o_j)$ 如果 o_i 和 o_j 能并行执行则为 1, 否则为 0。

该式的第一项支持合并那些部分实现可以共享的操作,这样可以减小硬件尺寸。比如加法和减法的实现相似,因为将加法器的一个输入求反就可以得到减法器。第二项支持合并共享数据的操作,这可以减小布线面积。第三项避免合并可以并发执行的操作,这可以提高系统的性能,因为将这样的两个操作合并在一起后就只能顺序执行,必然会降低性能。

为了改进该式,可以取式中的每项在设计中的重要程度作为其权重,如第一项乘以完成操作 o_i 和 o_j 所需的功能单元的面积,再除以设计的总面积。这样更容易合并成较大的模块。比如合并结果为 1000mm^2 的功能单元和 10mm^2 的单元相比,显然前者对整个面积的影响更大。第三项乘以 o_i 或 o_j 在从头到尾的执行周期里执行的可能性,再除以周期里的平均步数。这样做的结果是不常使用的操作更容易被合并,并且顺序执行,即使这些操作可以并发执行。这种合并能减小硬件尺寸,并且性能也基本没有降低,因为不常使用的操作对整体性能的影响很小。

206

通过上述接近函数, BUD 建立起层次化结群树, 并通过一个目标函数来对结群树上的每一级划分进行评估, 该目标函数是面积和执行时间的加权和。

控制流程如下所示: 首先, 计算各对操作间的接近值; 其次, 建立一个结群树, 其中使用接近值的平均值作为新的层次化对象和其他对象间的接近值; 再次, 对树中的每一个切割所得的划分, 评估其面积和时间; 最后, 选择成本最低的划分, 通过高层次综合产生其输出结构。

3. 举例

图 6-8 中通过一个例子来说明 BUD 划分技术。BUD 将图 6-8a 中的输入行为转化为图 6-8b 中的 CDFG (实际上 CDFG 是另一种不同的形式, 但不同之处这里没有影响)。执行过程从图中上面的方框开始, 即把 $a + b$ 的值赋给 x , 如果 $a = b$ 的话 $cond = '1'$, 否则 $cond = '0'$ 。如果 $cond = '1'$, 那么第二个方框将执行, 否则结束。第二个方框将 $x - y < z$ 的比较结果送入 C 中。

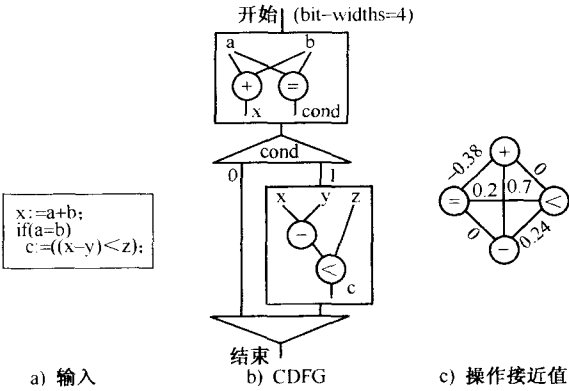


图 6-8 BUD 例子

图 6-8c 中给出了各个操作对之间的接近值, 计算过程如下。首先假设已知 FU_cost 表如下所示:

207

操 作	功 能 单 元	成 本
+	加法器	20
-	减法器	25
+, -	加法器/减法器	30
<	小于比较器	20
=	等于比较器	10
<, =, >	比较器	25
其他组合	—	成本和

因此, 加法操作所需的功能单元成本为 20, 而减法为 25。而将加法和减法合并到一个结群后, 成本仅为 30, 这比两个操作各自的成本和少, 这是因为加减法可以共享硬件。小于操作、等于比较操作以及小于/等于/大于比较操作的成本也均列于表中。其他操作组合后的成本均由组合的各操作成本求和而得。

根据 FU_cost 表, 并假定所有数据都是 4 位, 就可以计算出加法和其他操作之间的接近值:

概述

和 BUD 一样, Aparty 的输入是简单任务, 然后被转换为算法级数据流图。再将图中的操作划分为一系列的模块, 其中每个模块代表一个数据通路模块或是定制处理器(即控制器和数据通路)。这些模块中的每一个都代表一个芯片或者一部分芯片。模块的数量由划分算法决定。划分的输出定义了调度、分配和绑定, 可以作为高层次综合的输入。

划分中采用了多级结群算法。接近函数中考虑到的度量有下面三个目标。第一是减少控制传输(control transfer reduction), 就是结群之间传递控制信息的次数。假定划分能产生多个控制器, 那么对于结群之间通信慢的情况, 就可以通过这样的措施来提高性能。第二是减少数据传输(data transfer reduction), 即减少结群之间用于传输数据的连线, 这也可以间接地提高性能。第三是共享硬件(hardware sharing), 即通过共享功能单元来减少整个硬件大小。组合这三种类型的度量可以得到五个接近函数, 下面将分别对其进行说明。

操作的控制接近性(control closeness of operation): 这个函数的目的是减少控制传输。接近函数如下所示:

$$Closeness(o_i, o_j) = Prob(o_j | o_i) \quad (6-8)$$

式中 $Prob(o_i | o_j)$ 表示在操作 i 执行的前提下, 操作 j 执行的可能性, 其中两个操作同属于一个无环 CDFG 内。比如图 6-8 中, 有 $Prob(o_+, o_-) = 1$, 并通过模拟得到 $Prob(o_+, o_-) = 0.8$ 。这个函数的提出使得那些在一个无环行为内, 能够在一步里共同执行的操作更容易得到合并。

结群的数据接近性(data closeness of clusters): 这个函数的目的是减少数据传输。接近函数如下所示:

$$Closeness(p_i, p_j) = \frac{Conn(p_i, p_j)}{Total_conn(p_i) + Total_conn(p_j)} \quad (6-9)$$

式中 $Conn$ 和 $Total_conn$ 的定义同 6.5.2 节, 并从一对操作扩展到不同组之间。这个函数处理的是不同的结群而不是操作, 因此函数的形式是 $Closeness(p_i, p_j)$ 而不是 $Closeness(o_i, o_j)$ 。再以图 6-8 为例, 这个函数支持合并不同的结群, 否则在不同结群之间就需要许多数据线来传输数据。

结群的控制接近性(Control closeness of clusters): 这个函数的目的是减少控制传输。接近函数如下所示:

$$Closeness(p_i, p_j) = Prob(p_i \cap p_j) = Prob(p_i) \times Prob(p_j | p_i) \quad (6-10)$$

式中 $Prob(p_i)$ 是结群 p_i 中的操作被激活的可能性, 其中每个操作都属于 CDFG 的任一个无环中。这个函数倾向于合并可能在顺序行为中的同一步里执行的结群, 而不是操作。注意以下几点: (1) 这个函数考虑的是结群对而不仅仅是操作对。(2) 如果无环 CDFG 中含有循环, 附加的 $Prob(p_i)$ 因子就会提高共同执行操作间的接近值, 使其高于很少执行操作间的接近值。共同执行的操作对性能的影响较大, 因此希望它们能得到合并, 从而提高性能。(3) $Prob(p_i \cap p_j)$ 和 $Prob(p_j) \times Prob(p_i | p_j)$ 相等。这两个值可能并不相等, 因为过程间的调用并不一样。Aparty 使用这两个值之间的大者。

结群的参数数据接近性(parameter data closeness of clusters): 这个函数的目的是减少数据传输。接近函数如下所示:

$$Closeness(p_i, p_j) = \frac{CommCalls(p_i, p_j)}{\sum_k ExternCalls(p_i, p_k) + \sum_k ExternCalls(p_j, p_k)} \quad (6-11) \quad [211]$$

式中 $CommCalls(p_i, p_j)$ 是 p_i 和 p_j 共同调用的过程的数量; p_k 是一个过程, $ExternCalls(p_i, p_k)$ 是 p_i 中调用过程 p_k 的总次数, 如果没有调用则为 0。这个函数倾向于合并那些需要很多数据线来传递参数的结群。需要注意的是, 在一个公共函数被其他结群调用的时候, 式中的分母会减小接近性。相反如果某个过程仅仅被两个结群调用, 那么这两个结群之间的接近性会得到提高。

操作的功能单元共享性(functional unit sharability of operations): 这个函数的目的是提高硬件共享。接近函数如下所示:

$$Closeness(p_i, p_j) = \frac{\sum_{o_k \in p_i} Y(o_k, o_l) size(p_i) + \sum_{o_k \in p_j} Y(o_k, o_l) size(p_j)}{size(p_i) + size(p_j)} \quad (6-12)$$

其中各项意义如下所示:

$$Y(o_k, o_l) \quad \forall o_l \in p_j f(o_k, o_l) \wedge g(o_k, o_l),$$

$$f(o_k, o_l) \quad \text{如果 } o_k \text{ 和 } o_l \text{ 被调度在一个控制步内则为 1, 否则为 0,}$$

$$g(o_k, o_l) \quad \text{如果 } o_k \text{ 和 } o_l \text{ 能共享一个功能单元则为 1, 否则为 0.}$$

这个函数倾向于合并能够共享相同功能单元的操作, 而避免合并调度来并行操作, 否则这些操作将被重新调度来在一个功能单元上串行执行, 这显然会影响性能。注意: 这里的 CDFG 必须预先经过调度。

这些目标函数都可以用来选择最好的切割线。在 Aparty 中描述了三类目标函数, 但实际使用哪一个需要由设计者来指定。第一类目标函数为最小化面积。通过每个结群的功能单元最小数量, 以及对功能单元和多路选择器面积进行求和, 可以得到面积的评估值。第二类为最小化结群之间的连线数量。连线总数通过对每个结群访问外部数据的数量和大小求平均值而得。第三类为最小化调度长度。

对于 Aparty 内置的目标函数, 如果对面积或调度长度评估存在有多个有效切割线, 则选择评估值最高的一个。这里假设切割数越高, 就代表功能单元的共享性越强, 因此能减小总面积。

控制流程如下所示: 首先, 选择一个接近函数 $Closeness(o_i, o_j)$, 对于每一对对象, 计算 $Closeness(o_i, o_j)$ 。其次, 建立结群树, 通过最大值或最小值计算合并后新对象的接近性。再次, 选择目标函数, 对结群树中的切割线确定的划分进行计算。如果超出约束条件, 就对树的下一级切割线。最后选择成本最低的划分。如果设计者不结束算法, 则根据新的划分重复第一步。最终结果将作为高层次综合工具的输入。

设计者主要的交互体现在选择接近函数和目标函数, 以及指定面积和执行时间的约束。

6.5.4 其他技术

对于给定的算法级数据流图, [Geb92a] 中提供的技术能通过将问题转化为 ILP 来实现高层次综合中的调度、分配和绑定等任务。在 [Geb92b] 中将这种形式扩展到多芯片上, 这样图中的每一个操作都能映射到指定芯片的功能单元上。然后通过特定的技术来有效解决 ILP。求解 ILP 内容已超出了本书的范围。

[KP91] 中描述了一种名为 CHOP 的技术, 能够用来对数据流图的划分进行评估。对给定

划分中的每一组,这种技术首先产生一系列寄存器传输级的实现,并保证满足尺寸、引脚和时间等约束。这一步里的每一组都有数十或上百种可能实现方法,因此整个系统的实现方案可能成千上万。第二步需要在这些可能的系统实现中选取少量满足全局性能和位速率的方案。因此该技术可以应用到划分工具中,来实现对数据流图划分的快速评估。

6.6 软硬件划分算法

在嵌入式系统中经常将软硬件实现进行集成。软件在已有处理器上执行,可以减少制造成本,更容易进行修改,和相同功能的专用硬件相比能够更快完成设计。但硬件能够提供更好的性能。系统设计者的目标是使用尽可能少的专用硬件来完成整个系统,并满足需要的性能要求。换句话说,设计者应该尽可能使用软件来完成所需的功能。

软硬件划分问题是 6.4.1 节中定义常规划分问题的一种特例,其中的一个特点是二路划分,包括硬件和软件两部分。更重要的是划分中含有性能和硬件面积两个关键度量,通过将对象移到硬件上可以提高性能,反之则可以减小面积。这些特点使软硬件划分需要特定的算法。在这一节里,我们将讨论软硬件划分问题的三种推荐算法。

6.6.1 贪心算法

这是一种简单快速的算法,从一个初始划分开始,移动能够改善划分结果的对象,算法流程如下所示。其中的函数 $Move(P, o_i)$ 通过将对象 o_i 从软件移动到硬件(或从硬件移动到软件),返回一个新的划分 P' 。

算法 6.6.1: 贪心移动

repeat

$P_orig = P$

for i 从 1 到 n **loop**

if $Objfct(Move(P, o_i)) < Objfct(P)$ **then**

$P = Move(P, o_i)$

end if

end loop

until $P = P_orig$

算法 6.6.2 中为[GD92]中给出了贪心算法,它和算法 6.6.1 相似,并进行了扩展,以保证满足性能约束。算法中的函数 $Successors(o_i)$ 返回一组对象,它们都是系统功能内部模型中 o_i 的后继;函数 $SatisfiesPerformance(P)$ 在划分 P 满足各项性能约束的时候返回真值。

算法 6.6.2: Vulcan II 算法

/* 全硬件的初始划分 */

$P = \{O, \phi\}$

repeat

$P_orig = P$

for 硬件中的每一个 o_i **loop**

$AttemptMove(P, o_i)$

end loop

until $P = P_orig$

```

procedure AttemptMove(P,  $o_i$ )
  if SatisfiesPerformance(Move(P,  $o_i$ ))
    and Objfct(Move(P,  $o_i$ )) < Objfct(P) then
      P = Move(P,  $o_i$ )
    for Successors( $o_i$ )中的每一个  $o_j$  loop
      AttemptMove(P,  $o_j$ )
    end loop
  end if

```

215

该算法开始生成一个全硬件的划分方案,只要存在性能满足的划分方案,这种初始划分就能满足性能约束(实际上,某些不考虑约束的行为初始分配到软件上)。性能满足的划分就是一个满足所有性能约束的划分。算法中的移动对象,不仅要使成本有改善,而且还要满足各项性能(实际上这需要满足组间的最大接口约束)。一旦行为进行了移动,算法将试着先移动与该对象接近的对象。

贪心算法计算速度快,但其主要缺点是无法逃出局部最小点。

6.6.2 爬山算法

为了克服贪心算法的局限,研究者提出了爬山算法,如模拟退火算法。爬山算法能够接受负方向的移动,因此可以逃出多数局部最小点。只需要建立一个初始划分,然后应用该算法即可。

在[EHB94]中介绍了一种方法,它使用全软件划分方案为初始划分。然后使用爬山划分算法来提取对象从软件移动到硬件,以满足性能的约束。这种提取也可能将对象从硬件移动到软件,从而减小硬件的使用。

成本函数

下面来考虑爬山划分算法中需要使用的成本函数,其难点在于成本函数必须考虑性能约束和最小化硬件两方面因素,需要对其在成本函数中取得平衡。算法 6.6.2 中的成本函数不含有性能约束项,从而回避了这个问题;算法对不满足性能的所有划分都拒绝。显然这种方法很容易陷入局部最小。某些爬山算法在计算之前固定硬件尺寸,并在成本函数中去掉该因素来避免这个问题。这种方法的局限在于需要设计者手动尝试多种硬件尺寸,且每次都需要进行划分,最终才能找到满足性能约束且面积最小硬件尺寸的划分。

216

第三种解决方案是在成本函数中考虑这两方面因素,一项为所有性能不满足的和,另一项为硬件尺寸。为保证找到的解满足性能,性能项需要较大的权重;即最小化硬件尺寸应次要考虑。成本函数如下所示:

$$\text{Objfct}(P) = k_{\text{perf}} \times \sum_{i=1}^m \text{Violation}(C_i) + k_{\text{area}} \times \text{Size}(\text{hardware}) \quad (6-13)$$

式中 $\text{Violation}(C_i) = \text{Performance}(G_i) - V_i$, 如果计算结果为负则取 0。另外 $k_{\text{perf}} \gg k_{\text{area}}$, 但 k_{perf} 不能无限大, 否则对一个基本满足约束的划分和一个超出约束很多的划分, 算法无法对其进行区分。

这种算法也叫做性能权重爬山算法(performance-weighted hill-climbing, PWHC)。和算法 6.6.2 相比,它能够得到更好的结果。

6.6.3 二分约束搜索算法

PWHC 在成本函数中考虑了性能和硬件尺寸两方面因素,这种情况下通常能获得比贪心算法更好的结果。第三种算法可能获得更小的硬件,这种方法在一定程度上包括了将满足性能问题从最小化硬件中分离出来。软硬件划分问题变为确定最小硬件尺寸约束,而这个问题可能通过划分算法找到零成本的解。换句话说,必须在硬件尺寸从零到全硬件实现的范围内,寻找第一个零成本的解。在搜索中,第一个零成本的解应该接近最小硬件尺寸解。很显然,这类在序列中搜索某项的问题能够通过二分搜索来有效求解。

下面来讨论最小化硬件划分算法,该算法基于可能硬件约束范围中成本序列的二分搜索,也叫做 BCS(二分约束搜索)算法。算法中使用变量 *low* 和 *high* 来表示当前搜索步骤中,零成本约束所处的范围;变量 *mid* 表示这个范围的中间值。另一个变量 *P_zero* 保存搜索过程中,具有最小硬件约束的零成本划分。函数 *PartAlg* 代表迭代划分算法,如模拟退火算法。

算法 6.6.3: 二分约束搜索(BCS)软/硬件划分

```
low = 0, high = AllHwSize
while low < high loop
    mid = (low + high + 1) / 2
    P' = PartAlg(P, C, mid, Cost())
    if Cost(P', C, mid) = 0 then
        high = mid - 1
        P_zero = P'
    else
        low = mid
    end if
end loop
return P_zero
```

该算法在可能约束范围内,使用二分搜索策略,并对每个被搜索的约束都应用划分算法和成本函数进行判断。算法在标准二分搜索算法的基础上做了两点改动。其一是变量 *mid* 用来表示划分中的硬件约束,划分的结果用来确定成本,而标准算法中的 *mid* 为数组元素的序号。其二是成本与 0 进行比较,而在标准算法中数组元素与关键码进行比较。

这个算法仅是软硬件划分问题中的一部分,对普遍问题的研究还处于初级阶段。[TAS93]和[KL93]对该问题进行了综述,并讨论了诸如粒度、评估和软硬件协同模拟等问题。[EHB94]和[GD93]介绍了软硬件划分系统、算法以及协同模拟技术。在[YEBH93]中提到了一种快速软件性能评估技术,它能够很快对划分进行评估。

6.7 系统功能划分

至今已公布了许多在软硬件组件间对系统功能进行划分的技术,下面将对其进行简短的概述。对每种技术都将介绍其研究动机,然后讨论在 6.3 节中提到的相关问题。

6.7.1 Vulcan

1. 研究动机

Vulcan 由两部分组成。Vulcan I 在只有一个 ASIC 很小的情况下[GD90],在多个 ASIC

之间进行功能划分;而 Vulcan II 在软硬件之间进行划分,从而可以通过使用软件来减少 ASIC 的成本[GD93]。

2. Vulcan I 概述

Vulcan I 的输入是一系列任务和一系列芯片。划分的粒度可以是任务本身,也可以是语句块或者算法操作。Vulcan I 将输入的任务转化为层次化 CDFG。

Vulcan I 首先试着用粗粒度进行划分。如果找不到满足约束的划分,CDFG 的结点将被分解为较细的粒度,再重复进行划分。

Vulcan I 支持成组移动和模拟退火算法。它使用下面的目标函数,其中 $avgcut$ 是所有 p_i 切割数的平均值, T 是调度长度, $maxschedule$ 是允许的最大调度长度, k_1 和 k_2 是常数:

$$Objfct = k_1(avgcut) + k_2(T - maxschedule)$$

不满足面积、引脚或执行时间约束的划分都被认为是无效划分。

Vulcan I 输出 CDFG 和每个结点映射到芯片的情况。CDFG 和映射信息作为高层次综合工具的输入,分别对每个芯片的硬件进行综合。

控制流程如下:首先合并将在相同硬件上执行的操作,保证这些操作在划分的时候不被分开;其次对每个操作的面积、每个互联的线宽和数据流图的调度长度进行评估;再次调用二路划分算法。如果结果无法满足面积约束,则将最大操作的子图进行分解,然后对该子图进行划分。重复调用二路划分算法来实现多路划分;最后,对每个芯片的结构进行综合。

3. Vulcan II 概述

Vulcan II 将输入的一系列任务分解为语句块,再进行划分。它假定目标体系结构具有一个处理器、一个可被分为多个芯片的硬件组件、一个系统总线和一個公共存储器,所有的软硬件通信都通过这个存储器来完成。

这种技术的关键点在于将操作分为三类。具有无限时延的操作称为不确定操作。这种不确定性如果由等待外部事件而引起,该操作即为外部不确定操作;如果是由内部数据的依赖关系引起的,比如依赖于数据的循环条件,那么该操作被称为是内部不确定操作。所有其他的操作时延有限,因此都是确定操作。首先假设内部不确定操作不能受性能的约束,因此这些操作都安排在软件上;以及所有外部不确定操作必须安排在硬件上。因此,只需要对剩下的确定性操作进行划分。

Vulcan II 中使用的目标函数包含有硬件尺寸、程序/数据储藏量、总线带宽、数据传输速率、同步开销和特定操作间的时间等多种度量。它使用 6.6.1 节中提到的定制硬件/软件划分算法。开始划分中所有确定性操作都在硬件上,算法在满足性能约束的前提下,将操作移动到软件上,从而减小硬件成本。结果划分将作为高层次综合和编译工具的输入。

6.7.2 Cosyma

1. 研究动机

[EHB94]中提到的 Cosyma 技术对输入的一组任务,能够自动地在简单的软硬件体系结构上实现划分,这和 Vulcan II 相同。该技术基于软件实现,表现在(1)输入的可执行描述可以包含指针之类的结构,这些只能在软件上实现,(2)而且划分算法在满足性能约束的前提下会将尽可能多的功能划分到软件上。

2. 概述

Cosyma 将输入的任务分解为语句块,再进行划分。它使用和 Vulcan II 类似的目标体系

[219]

[220]

结构,其中含有一个处理器、一个硬件组件、一个公共存储器和一个系统总线,所有的软硬件通信都在系统总线上完成。

Cosyma 使用的目标函数中结合了语句块执行时间和通信时间。Cosyma 使用模拟退火划分算法,而且可以增加其他算法。它从全软件初始划分开始,算法将对象移动到硬件上,直到满足性能约束为止。

221 这种技术的关键点在于在划分和实现之间进行迭代。在划分后,设计者可以使用 C 编译器和高层次综合工具来分别实现软硬件部分,然后对设计进行模拟,以及评估划分方案。根据划分结果,设计者再重新进行划分。未来的发展会将评估结果直接合并到目标函数中,从而实现自动重新划分。

6.7.3 SpecSyn

1. 研究动机

在 6.1 节中提到了系统设计中的两个主要问题,分配物理系统组件和在组件间划分行为。上面描述的划分技术主要是针对对逻辑块、功能单元、芯片或处理器间划分功能进行讨论。SpecSyn[G VN94, VG92]从三个方面来对该问题进行了扩展。第一是包含了存储器和总线这两种新的系统组件,这二者在系统设计中很重要,但前面提到的技术都没有考虑。第二是包含了变量和通信通道两方面的功能,它们和行为共同组成了可执行描述,但也经常被忽略。第三是将物理系统组件的不同数量和类型的划分看做系统设计的一部分。

在 SpecSyn 中有三个很明显的划分问题。第一是将行为映射到系统组件上,如定制或标准处理器;第二是将变量映射到存储器上;第三是将通信映射到总线上。其他划分技术里通常都忽略了后两个问题。

222 另外,在执行划分任务的时候,SpecSyn 提供了较好的交互式功能。这些交互也是系统设计工具的关键部分。在其他技术中,由于划分的对象和设计者关系不大,因此与设计者的交互很有限。而 SpecSyn 输出的是易读的、可修改的精确的描述,而不是一个完整的实现。这样的描述支持手动和自动组件设计技术的共同执行。

2. 概述

SpecSyn 划分器将输入的任务集分解为三种不同的粒度。第一种粒度就是任务本身,第二种包含了任务和子例程,第三种为语句块。设计者可以对输入的不同部分选择不同的粒度。SpecSyn 支持多种系统组件,包括芯片、片上块,现成的处理器、存储器(用于变量)和总线(用于通信)。它支持的度量包括芯片面积、块面积、任务在软硬件上的执行时间、软件指令和数据大小、存储器大小、总线宽度、位速率和成本等。目标函数为各种约束冲突的加权和,并且可以由设计者增加其他成本函数。

如上所述,设计者必须解决划分中的三个问题。由于解决这些问题不存在最佳顺序,设计者可以以任何顺序来处理,并可以对其重复求解。任何基本划分算法都可以解决每个划分问题,虽然每个问题都需要不同的接近度量和目标函数,但算法控制策略都不变。SpecSyn 使用特殊的软硬件划分算法,即 PWHC 和 BCS(见 6.6 节)。下面将对 SpecSyn 支持的几种接近度量进行介绍。

3. 定制/标准处理器行为

将行为合并后在定制或标准处理器上执行,由此可以引出四个接近度量。互联(intercon-

nection)是行为间共享的连线数的估计值。合并互联很多的行为可以减少引脚数。通信(communication)是行为间数据传输位数据位数的评估值,它和用于传输数据的连线数相互独立。合并通信较多的行为可以减少通信时间,从而得到较好的性能。顺序执行(sequential execution)为布尔值,当某个行为为了完成其正确功能而必须顺序执行的时候取真值。将顺序执行的行为映射到同一个控制器上不会降低性能,而将可以并发执行的行为映射到同一个控制器上则会降低性能。硬件共享(hardware sharing)是行为间能共享的硬件所占百分比的估计。将能共享硬件的行为进行合并可以减小整个硬件尺寸。

223

4. 变量映射到存储器

为了描述对变量组合并在存储器上实现,有三个接近度量。顺序访问(sequential access)是布尔值,当变量只能顺序访问的时候为真。将顺序访问的变量映射到同一存储器上不会降低性能,而将可以并行访问的变量映射到同一存储器上则会由于访问冲突而降低性能。共同访问量(common accessors)是访问所有给定变量的行为数。将这些变量进行合并可以减少总连线数。宽度相似度(width similarity)是变量之间位宽的相似度。将位宽相似的变量合并可以减少存储器位宽的浪费。

5. 通道映射到总线

为了描述通道的组合并在总线上实现,有三个接近度量。顺序访问是布尔值,当通道只能顺序访问的时候为真。将顺序访问的通道映射到同一总线上不会降低性能,而将可以并行访问的通道映射到同一总线上则会由于访问冲突而降低性能。文献[FKCD93]中对合并顺序访问通道计进行了讨论。共同访问量是访问给定通道的行为数量。将这些通道进行合并可以减少总体连线数。宽度相似度是通道之间位宽的相似度。将位宽相似的通道合并可以减少总线宽度的浪费。

SpecSyn 划分器的输出信息为一组映射关系,可以作为细化工具的输入。细化工具产生新的描述,功能对象在最近介绍的系统组件上进行划分。如果需要,工具还将增加通信细节和仲裁器。

SpecSyn 的控制流程和设计者的交互将在第 9 章里进行详细讨论,这里仅对其指示和反馈做简单概括。主要指示是在系统组件上对功能对象进行手动划分。因为多数功能对象能被设计者所理解,使这种手动的交互成为可能。设计者可以手动划分系统组件。第二种指向是设计者对目标函数进行定制。最后是设计者能任意选择自动算法执行的顺序。给设计者的主要反馈信息是不同设计度量的评估值,以及这些值和给定约束之间的比较结果。另外,功能对象之间的接近性信息也能提供给设计者。

224

6.7.4 其他技术

给定一系列任务和一组互联的处理器,一个普遍的问题是将每个任务的执行调度到指定的处理器上,以满足性能的约束。文献[SP91]中提出的一种技术将这个问题进行扩充,考虑了处理器的选择,而不是仅提供互联的处理器作为输入。此时输入为任务和任务间的数据通信,其中任务的内部功能不需要指定。另外,需要给出各种可选的处理器类型、成本等参数,以及每个任务在各类处理器上的执行时间。处理器分配和任务调度问题可以用一个 ILP 的形式来描述。设计者的交互包括了给 ILP 增加约束。

6.8 折中的探索

由可执行描述进行功能划分,使得对设计空间进行快速搜索成为可能。为了说明这点,来看下面这个系统组件的分配问题。给定一些系统组件的类型,必须选择各类以及许多的组件来使用。这种选择比较困难,因为要在划分实现后才能得出分配中的质量度量值,而划分实现又需要花费很多时间。通过评估可以比完全实现能更快地获得这些质量度量。因为对于可执行描述方法的划分和评估可以自动完成,这些度量也就能很快获得。

下面用一个例子来说明在权衡时如何快速获得度量值。这是医学中用来实时测量膀胱体积的例子。系统外部的监控设备输入声纳数据,然后系统计算这些数据,来检查膀胱壁及膀胱的体积,这需要能动态调整监控设备的位置。

假设有三个厂家,各自提供一种芯片组,即系统组件的成本和性能等特性都不相同。我们必须选择一种芯片组,然后将功能分配到芯片组上。另外,假设处理器在三种芯片组上都可以使用。

首先对每个芯片组,都自动产生所有可能的分配方案,并且将成本上界定为 145 美元。对于每种分配方案,都对系统功能进行自动划分,并考虑性能和封装大小、引脚等约束。如果不满足大小和引脚约束,就不能使用该分配方案。对剩下的分配再对系统性能进行评估。对这个例子,整个自动化过程在 Sparc 2 上需要 2.5 小时的时间。

图 6-10 中给出了计算结果,其中每条曲线的第一个点对应全软件设计,即所有功能都在处理器上完成。剩下各点对应其他分配方案。芯片组类型各自不同,有的包含处理器,有的没有处理器。假设我们想要成本最低且性能最好的结果,在图中可以看到芯片组 1 提供的解(点 A),它对应一个具有 20 000 门的简单芯片。芯片组 2 也提供了相近性能的解(点 B),但成本略高,它也使用了 20 000 门的芯片。使用芯片组 3 来达到相近性能的成本更加贵,如点 C 所示。如果不需要性能最佳,而只要求性能在 650 微秒以下即可,那么芯片组 1 提供了最便宜的解(点 D),它含有一个处理器,以及 10 000 门的芯片。

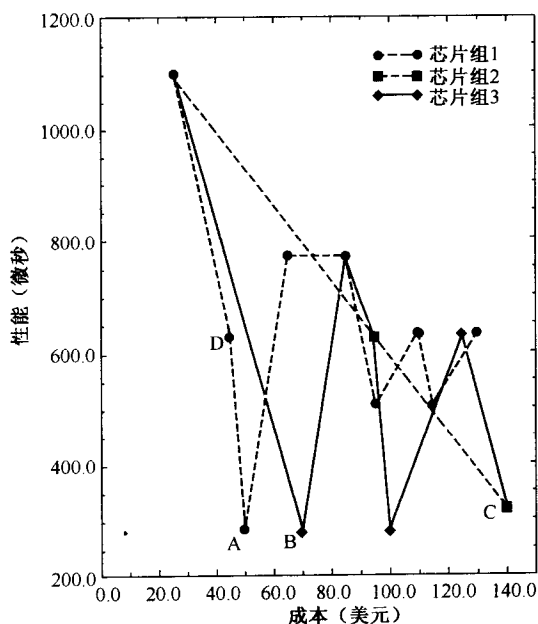


图 6-10 体积测量系统中,三种芯片组的成本和性能平衡分配

对于选择分配方案,有着许多需要考虑的因素,包括成本和系统性能。一次性工程成本(NRE)、功耗、封装尺寸、开发时间、生产、厂家的可靠性和厂家的支持等。对比不同分配方案,在这些因素之间取得平衡是一件复杂的工作,对此折中曲线能发挥很大的作用。

6.9 结论和发展方向

功能划分是系统设计中的一个关键部分,它对最终设计的质量有着很大的影响。功能划分由可执行描述开始能获得最好的结果,因为这种描述容易被机器所理解,可以在划分中使用自动评估和划分工具。这些工具允许对不同设计进行快速搜索,从而获得很好的总体设计。还有一个优点在于可执行描述还可以和工具中的某些特定步骤相结合,共同形成优秀的设计文档,这能够极大地减少设计和再设计时间。

未来的研究有多个主题。在设计完成实现后,划分中的评估可能不够准确,还可以对设计进行改善。如果在实现中能获得度量,来作为划分任务的反馈,那么重新划分的时候就能够用到这些更为精确的度量。换句话说,实现中获得的度量值越精确,划分算法就越容易找到更好的解。

对不同划分算法进行比较分析也有很大作用。现在已有一些对电路划分进行比较的算法,但其对功能划分并不适用。此外值得关注的问题是,通过做出多种可能选择以及对接近度量进行排序,所得到划分结果的质量。

粒度对划分质量也有影响,这点不难理解。如果能在多种级别粒度上进行划分,那有望得到较好的结果。

许多系统都表现得很规则。这里的规则指系统中的许多行为都彼此一样,仅仅是行为处理的数据有所不同。这种系统通常用多个相同的芯片来设计;因此只需要对一个芯片的设计进行实现。将来的算法会将划分规则和半规则行为的技术结合起来。

总之,由于划分已经比较成熟,未来主要的任务是对已有的技术进行调整,并将其应用到功能级。

227

6.10 练习

1. 对本章中提到的每个算法,计算各自的复杂度。
2. (a)将 N 个对象划分成 M 部分,求划分的映射数量。(b)将 100 个对象划分成 3 部分,有多少种可能的映射方式?(c)如果一种划分算法检测 N^2 种划分,那么它能检测所有可能划分的多少?
3. 对第 4 章中的应答机,将其行为 Answer 划分为下列粒度:(1)任务级,(2)语句块,(3)语句,并列举出各种情况的功能对象。为简单起见,忽略行为 *RemoteOperation*。(a)每种情况各有多少个对象?(b)哪种粒度可能进行手动划分?
4. 假设一个可执行描述可以被分解为 50 个行为、250 个语句块、1750 个语句或者 2000 个控制/数据流图的操作。如果某个划分算法检验 n^2 个划分,对每个划分需要 5 毫秒来进行评估,那么对这四种分解粒度,该算法分别需要的时间大致为多少?
5. 说明层次化结群和多级结群的区别,以及各自的优缺点。
6. 对图 6-11 中的例子,计算各对对象间的接近值,使用 BUD 中提出的接近函数,其中加法器和减法器成本均为 1。

228

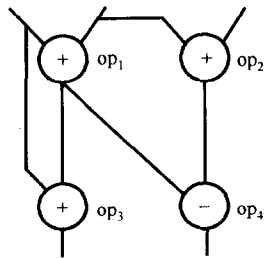


图 6-11 数据流图例子

7. 对第4章中的应答机模型,手动完成行为 *Answer* 的结构和二路功能划分,目标为基本尺寸相等的划分,并使引脚数最少。在评估尺寸和引脚两方面对各种方法的最后实现结构进行对比。有余力者可以对整个系统再做一遍。
8. 如图 6-12,说明成组移动是如何改善划分的。图中每个顶点大小均为 1,每条边权重均为 1。假设目标函数为 $5 \times \text{SizeViolations} + \text{Edges_crossing}$,其中划分的尺寸约束为 4。以字母顺序来执行移动,如果多次移动的成本相等则取第一个。注意,在获得全局最小成本以前,可以接受负方向的移动。
9. 对成组移动进行扩充,使其对图 6-13 中的例子,能够优先选择 *E* 而不是 *D* 进行移动,因为 *E* 的后续移动对象 *C* 可以减少一条边的交叉。提示,参见文献[Kri84]。

229

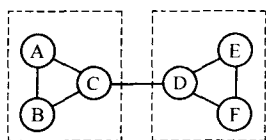


图 6-12 划分例子

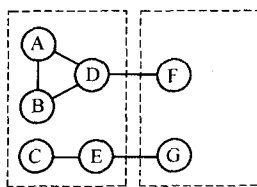


图 6-13 划分例子

10. 对一个图划分成 M 个部分的划分问题,求其 ILP 形式,给定的约束是总体尺寸和每个划分部分的切割数。进一步讨论如何对这种形式进行扩充,使其适用于功能划分。
11. 假设在 CDFG 操作级进行软硬件划分。CDFG 的一部分将映射到软件上。(a)设计一个算法,来将 CDFG 转换为相同功能的软件,如图 6-8b 中的例子。(b)使用该算法,将图 6-8b 中的 CDFG 转换为软件。
12. 考虑图 6-8b 中的 CDFG。假设操作 $+$ 和 $-$ 都映射到硬件,其他部分映射到软件。对软硬件部分分别给出 VHDL 的功能描述,注意要保证这两个组件接口的正确性,以及完成通信的具体方式。
- * 13. 将本章中提到的算法结合在一起,实现一个带权图的划分工具。
- * 14. 使用上文中的工具,选择一些例子,对不同算法进行比较分析,比较其运行时间和设计质量,其中目标函数自选。
- ** 15. 设计一种方法,来将质量度量合并到划分中,以此来鼓励带评估的划分步骤和实现步骤间的交互。
- ** 16. 设计一种方法,将功能划分和高层次综合结合起来。

230

231

第7章 设计质量评估

在上一章中,我们讲述了有关将系统功能在各组件间进行划分,使诸如性能和面积等设计度量的约束得以满足的一些基本问题和技术。然而,为确认这些约束是否已真正满足,我们必须能尽快地获得针对某些度量的评估结果。在本章,将讲述一些可用于对软/硬件的质量度量进行快速评估的一些技术。

7.1 引言

在系统级对设计质量进行评估具有本质上的重要性。有两个原因:第一,使设计者能将任何设计度量的估值和对该度量的约束值进行比较,以此对设计质量进行评估。例如,如果对某设计的估计面积大于所允许的芯片面积,则设计描述中的某个或某些数组变量就需要用芯片外的存储器实现,以满足芯片尺寸约束。第二,使设计者凭借对设计决策的快速反馈,对各种设计选择进行探索。通过这种方式,设计者能探索大量的设计选择,而不是对每个设计选择都是先综合成完整的实现,再测定特定的设计质量度量。

我们可以定义一个设计模型(design model)用于对质量度量进行评估。任何评估的有效性和可用性取决于该设计模型对真实设计的表示精度。

233

为表明设计模型对评估的影响,在图 7-1 中列出了具有代表性的几种设计模型,这些设计模型可用于根据系统的功能描述来评估系统的面积。图中各种类型的设计模型体现出为提高面积评估的精确性而需要的细节进展(progression of detail)。针对图中的每个模型,我们也列出了评估工具必须执行的其他一些任务,以达到比前面模型更精确的评估。

	设计模型	额外的任务	精确性	保真性	速度
a)	Mem	Mem分配	低	低	快
b)	Mem+FUs	FU分配	↓	↓	↑
c)	Mem+FUs+Reg	生存期分析	↓	↓	↑
d)	Mem+FUs+Reg+Muxes	FU绑定	↓	↓	↑
e)	Mem+FUs+Reg+Muxes+Wiring	布图规划	高	高	慢

Mem: 存储器 FUs: 功能单元 Reg: 寄存器 Muxes: 多路选择器

图 7-1 典型的评估模型及其精确性、保真性及评估速度

例如,如果仅仅把系统中存储器大小作为一个模型,如图 7-1a 所示,就需要进行存储分配(memory allocation)以确定存储器的类型和大小,乃至设计面积。如果在模型中加入功能单元,如图 7-1b 所示,则除了存储器分配之外,还需要进行功能单元分配(functional unit allocation)以确定设计中功能单元的类型和数量。类似地,若在模型中结合进寄存器和锁存器,如图 7-1c 所示,则需要进行变量的生存期分析(lifetime analysis),以确定在各个时刻需要存储的变量数。进一步在模型中加入多路选择器,如图 7-1d 所示,则需要进行功能单元绑定(func-

234

tional-unit binding),以确定存储单元到功能单元的连接数量,并以此确定多路选择器的大小。最后,如图 7-1e 所示,若在设计模型中结合进布线的面积,就需要进行布图规划(floorplanning)以确定组件的实际位置和方位,并以此确定它们之间互联线的大致位置和长度。

7.1.1 精确性与速度

前面的讨论使我们明白了在设计模型选择中的一个重要问题,即在评估的准确性与评估计算速度之间取得折中。评估的准确性(accuracy)是一种度量,它度量的是估计值与设计实现后经过测量所得到的实际值之间的接近程度。设 $E(D)$ 和 $M(D)$ 分别表示评估值和经过一种设计实现 D 后针对某种质量度量的真实测量值,则评估的精确度 A 定义如下:

$$A = 1 - \frac{|E(D) - M(D)|}{M(D)} \quad (7-1)$$

依据此定义,一个理想评估的精确度 $A = 1$,其他非理想评估的精确度小于 1。精确度依赖于设计模型所呈现细节的程度。例如,图 7-1a 中的模型用存储器面积的总和来近似设计面积。基于这种简单模型的评估工具运行速度快并且易于建立,其原因是在评估度量的计算中,仅需要考虑少部分设计特性。然而,简单模型精确度较低,而这种精确度正是引导系统设计者做出正确设计选择所需要的。

另一方面,一个设计模型很可能结合了设计的若干方面。比如,为了进行详细的面积评估,除了要考虑布线面积外,还要确定存储器、功能单元、寄存器以及多路选择器的数量和大
235 小。这些考虑还将进一步引起诸如功能单元分配、变量生存期分析、功能单元绑定以及布图规划等任务的执行,如图 7-1e 的例子所示。基于这种详细设计模型的评估工具需要更长的计算时间,但同时也会得到更精确的评估,因而也有助于做出一个更好的设计选择。

一般来说,采用简化的评估模型将获得快速的评估工具,但同时也导致更多的评估错误和较低的精确度。另一方面,只要质量度量的评估值能保证系统设计者在任何两个设计选择之间做出适当的折中决策,则不一定总需要很高的精确度。在这种情况下,我们就需要选择一个能产生高保真度估值的评估模型。

7.1.2 评估的保真度

一个评估方法的保真度(fidelity)[KGRC93]定义为在设计实现之间正确地预测了其对比关系的百分率。对于两个不同的设计实现,将某设计度量的评估值与测量值相比,如果两者具有相同的对比关系,则该评估正确地比较了这两种设计实现。换句话说,评估度量可用来在许多可能的设计实现中做出最佳选择。评估的保真度越高,就越有可能依靠对两种设计实现的评估,做出正确的设计决策。

为了对保真度做出正式定义,我们来看一个任意的功能描述。假设 $D = \{D_1, D_2, \dots, D_n\}$ 为该描述的一组设计实现。若对于所有的 i, j , 有 $1 \leq i, j \leq n$ 及 $i \neq j$, 则 μ_{ij} 定义如下:

$$\mu_{ij} = \begin{cases} 1 & \text{若 } E(D_i) > E(D_j) \text{ 且 } M(D_i) > M(D_j) \text{ 或者} \\ & E(D_i) < E(D_j) \text{ 且 } M(D_i) < M(D_j) \text{ 或者} \\ & E(D_i) = E(D_j) \text{ 且 } M(D_i) = M(D_j) \\ 0 & \text{其他} \end{cases} \quad (7-2)$$

评估方法的保真度 F 则可定义为正确预测的百分率,如下式所示:

$$F = 100 \times \frac{2}{n(n-1)} \sum_{i=1}^n \sum_{j=i+1}^n \mu_{ij} \quad (7-3)$$

图 7-2 显示出对某质量度量的评估保真度计算,这些计算是基于 A、B、C 三个设计点。首先来看图 7-2a 所示的由评估工具产生的对于某质量度量的评估结果和从设计实现获得的实际值。对于该度量,设计实现 A 的评估值和实际值都比设计实现 B 的要高,对于设计实现点 (B,C) 和 (A,C) 也有这种评估值和实际值的相对关系。因此,该评估工具的保真度为 100%。

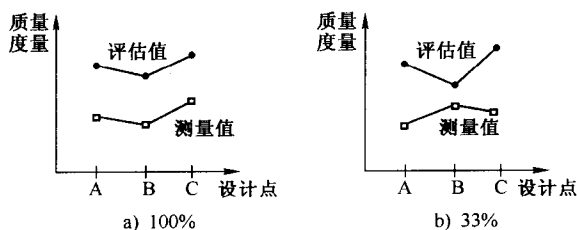


图 7-2 评估保真度的两个例子

现在再来看图 7-2b 中的设计点对 (A,B), 我们会发现当 $M(A) < M(B)$ 时却有 $E(A) > E(B)$ 。对于设计点对 (B,C) 也有这种相对关系。在这三个设计点对中, 由公式 7-2 可以计算出, 只有 (A,C) 有非零的 μ_{ij} 值。因此, 图 7-2b 中评估工具的保真度仅为 33%。

当从若干种设计实现中进行选择时, 基于高保真度的评估所引发的设计质量预测, 一般会引导更好的设计。保真度依赖于用于评估设计参数的设计模型。一般来说, 模型越精确, 评估保真度就越高。

237

7.2 质量度量

在本节中, 我们将讲述常用于表征设计的一些质量度量。其中两个最重要的度量为硬件和软件实现的成本和性能。虽然其他度量也非常重要, 但许多高层次决策是完全基于这两个度量的。

7.2.1 硬件成本度量

硬件成本度量包括制造芯片的成本、封装成本、测试成本以及分摊的工程与设计成本。

制造成本取决于设计实现的大小。最常用的尺寸度量是设计面积, 它是一个设计实现所需要的硅面积。对于一个给定的设计, 面积评估一般包括存储单元 (寄存器和存储器)、功能单元 (ALU)、互联单元 (多路转换器和总线)、控制逻辑等所需要的面积以及连接这些组件所需要的布线面积。

系统级的面积评估有助于判定设计的某个部分是否适于安置在芯片上的一个给定区域, 该区域在制造过程中预计会产生最大的成品率。例如, 如果某设计经评估预计有 $80\,000\ \mu\text{m}^2$ 的面积, 那么它将不能在最多能提供 $50\,000\ \mu\text{m}^2$ 面积的芯片上实现。必须将该设计划分, 并在两个或多个芯片上实现。

有时, 设计实现的尺寸不是用设计面积来近似, 而是用相关的度量, 如晶体管、门、寄存器级元件等的数量, PC 板的大小, 或者系统所需要的机壳空间。例如, 对于半定制 (门阵列) 和可编程 (FPGA) 技术, 每个芯片上的门的数量或者组合逻辑块的数量是有限的。对这些度量的评估将使系统设计者能够确定需要多少门阵列或 FPGA 来实现设计。这种尺寸度量具有很

[238] 高的保真度,因为每个晶体管、门或寄存器级元件的大小是已知的,所以系统面积就近似于这些元件大小的总和乘以一个适当的常数,该常数考虑的是布线和输入/输出的因素。此外,在一个设计周期中,可利用的组件数目要比可利用的芯片面积更早预先知道。

封装成本常近似为设计中引脚的数量。对引脚的评估能力是很重要的;设计者总是努力减少设计中引脚的数量,其原因是芯片上引脚的数量不仅影响封装成本,而且由于需要连接板上的芯片,从而影响电路板面积。

7.2.2 软件成本度量

虽然硬件实现具有性能高的优势,但也有部分设计用软件实现,运行于特定的微处理器上。软件实现具有若干优点。首先,实现成本非常低,因为处理器和微控制器等可编程器件通常都是大批量制造的。其次,用软件实现开发时间可大幅度缩短。给定一个可执行描述,软件实现的主要步骤是编译,而硬件实现则需要完成若干个设计任务,包括测试和制造。最后,软件实现有助于在设计后期对设计描述进行改动。而在硬件实现中,这种改动经常需要大量的重新设计。

用软件实现的设计描述被编译到所选择处理器的指令集。程序存储量与数据存储量是与软件实现相关的两个成本度量。程序存储量(program memory size)是用于存储由设计描述编译成的指令的存储量。数据存储量(data memory size)表示的是存储所有数据值(变量和数组)所需要的存储量,这些数据值由系统在执行计算的过程中产生并操作。例如,对于一个计算一组数据值的标准差的程序,且这些数据值存储于一个数组中,则设计者会关心的是:在特定的微处理器上,需要多少条指令,以及需要多少存储单元来保存该数组和所有中间值。

[239]

对程序与数据大小度量的评估是很重要的,它直接地影响了设计成本,并间接地决定了系统的性能。特别是,我们可能需要确定一个程序是否适合于一个处理器内置的程序存储能力。如果不适合,则必须增添额外的程序存储芯片,从而导致设计成本的增加。另外,我们可能需要了解数据量大小,因为它影响着数据存储单元的需要量,而这又反过来决定了数据存储的成本。我们可能还需要确定数据量是否小到可以存储于片上寄存器或高速缓冲存储器,以期得到更好的性能。

7.2.3 性能度量

性能度量可分为计算度量与通信度量。计算度量是对在一个行为中执行计算所需时间的度量。这种度量可根据在计算度量时所采用的时间单位类型来分类。一般可采用三种类型的单位:时钟周期、控制步数和执行时间;通信度量则与某行为与系统中其他行为进行相互作用所需时间相关。

[240]

1. 时钟周期

系统设计中的一个重要决策就是对系统实现的时钟周期进行选择。在进行任务综合之前选择时钟周期非常重要,因为该选择将影响到执行时间和实现系统所需资源。

例如,在图 7-3 中使用三种不同的时钟周期(380 ns, 150 ns, 80 ns)来实现同一个数据流图,图 7-3。在图 7-3a 中,时钟周期 380 ns 可使系统具有最快的执行时间,但需要使用 2 个乘法器和 4 个加法器实现,其中乘法器和加法器分别有 150 ns 和 80 ns 的时间时延;另一方面,图 7-3b 中的 150 ns 时钟周期仅需要一个加法器和一个乘法器,但它所需的执行时间为 600 ns。以每单位资源的执行时间为计,最有效的设计实现是采用 80 ns 的时钟周期,如图 7-3c 所示。其执行时间与第一种实现差距不大,但所需的资源数与第二种实现相同。

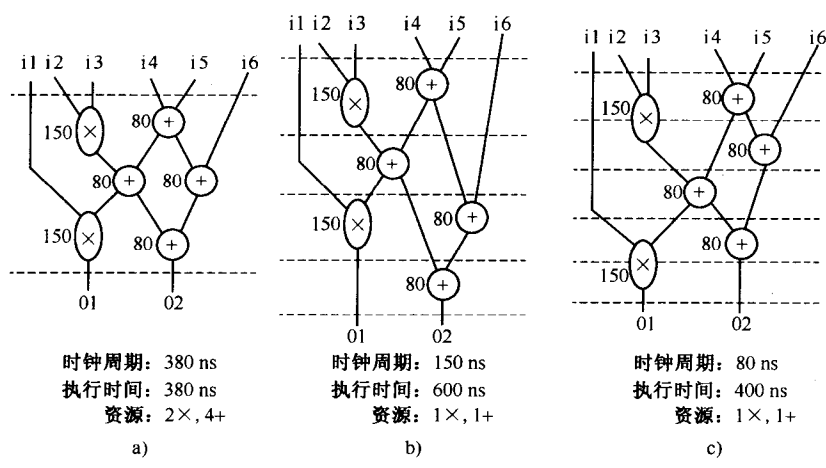


图 7-3 时钟周期对执行时间和所需资源的影响

时钟周期还决定了可用于实现设计的工艺。特定的工艺库规定了库中特定元件的最大工作频率。例如,VDP100 库[VTI88]规定 75 MHz 为最大频率,在该频率下双稳态电路的输入时钟可被驱动并保持稳定的逻辑电平转换。时钟频率高于最大规定值,则在设计实现中就不能采用该库。

2. 控制步

2.9.3 节中介绍的 FSMD 体系结构中,控制单元将系统中的操作安排为一系列控制步(control step)。一个控制步对应于控制单元状态机的一个单独状态。在综合过程中,功能描述中的操作被分配给这些控制步。实现一个行为所需的控制步数量影响到实现中控制逻辑的复杂性。如果一个设计经过调度确定为 N 个控制步,那么状态寄存器的位数则为 $\log_2 N$ 。另外,如果一个行为的代码为顺序执行的结构,那么该行为的执行时间则与控制步数成正比。即使行为中含有分支与迭代结构,控制步数也影响执行时间。比如,如果一个循环体有很多控制步,那么该循环体作为整体就具有比较长的执行时间。

241

3. 执行时间

设计中行为的执行时间定义为行为从开始到结束所需的平均时间。执行时间与执行该行为所需的控制步数成正比。

对执行时间进行评估的重要性体现在两个方面。首先,对某些行为可能已规定了性能约束,如图 7-4a 中的行为 B,它具有 10 ms 的最大执行时间约束。这个约束可能来自于实时要求,比如从旋转的磁盘中以固定的速率读取数据。设计者必须能估计出任何设计决策对受约束行为的执行时间的影响,以确定是否正在违反时间约束。例如,设计者可能想知道,通过分配两个加法器来实现行为 B 是否会违反性能约束。其次,执行时间约束将会对选择用于设计实现的工艺或元件库造成影响。例如,对执行时间的评估,将会使设计者能够在低成本标准处理器及其编译的软件和高成本定制的 ASIC 实现之间做出选择。

242

设计常常是通过流水线(pipeline)实现的。流水线的输入是以恒速率到达的数据流样本。流水线针对这些数据流进行计算,并以相同速率产生结果输出流。采用流水线的主要目的是提高计算的速率,提高设计中功能单元的利用率。实现第一个目标的方法是:将行为中的一系列计算分为几个阶段,并在每个阶段后将部分结果锁存起来;实现第二个目标的方法是:将具有

较大时延的功能单元分割成两个或更多个流水线阶段。流水线化功能单元允许对若干对操作数进行并发操作,对每对操作数所进行的操作都在同一流水线单元内得到某一阶段的部分计算结果。这样,时钟周期就缩短到等同于最长阶段的时延。

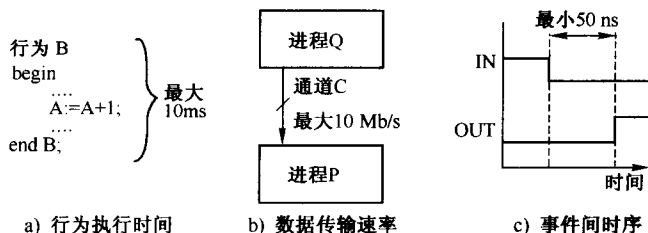


图 7-4 系统级性能度量

两个通常与流水线设计相关的度量是:阶段时延和执行时间。

流水线的阶段时延(stage delay)是指任何阶段执行其计算时所需要的时间。由于流水线中的所有阶段都从同一时刻开始处理数据,因此阶段时延也代表了流水线输入能接受两个连续数据样本的最小时间间隔。流水线的吞吐量(throughput)度量的是流水线产生出结果的频率。换句话说,如果 $stage_delay$ 表示流水线中最长阶段的时延,那么该流水线的吞吐量和阶段时延具有如下关系:

$$throughput = \frac{1}{stage_delay} \quad (7-4)$$

流水线的执行时间是指从数据到达流水线开始到产生相应结果之间的总时间。流水线的执行时间也常称为潜伏期(latency)[HP90]。若流水线阶段的数量用 num_stages 表示,则流水线执行时间可如下计算:

$$execution_time = num_stages \times stage_delay \quad (7-5)$$

图 7-5 显示出非流水线设计与流水线设计的区别。行为的数据流图如图 7-5a 所示。采用 2 阶段乘法器和锁存器(在图 7-5b 中用阴影矩形表示)可以将该数据流图划分成为 4 阶段流水线。图中也显示出流水线的阶段时延和执行时间。假设每个阶段的时延等于一个时钟周期,那么,对应于一组输入,在 4 个时钟周期之后,这两种情况会得到相同的结果。但是,对于图 7-5a 所示的具有四个时钟周期阶段时延的非流水线设计,其吞吐量是每 4 个时钟周期产生一个结果。另一方面,在图 7-5b 所示的流水线化设计中,具有一个时钟周期的阶段时延,其吞吐量将是每个时钟周期产生一个结果,也就是说,是非流水线设计的 4 倍。

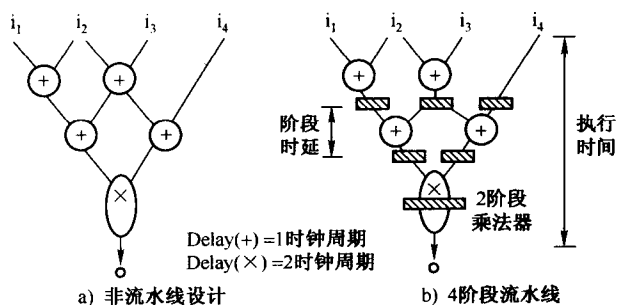


图 7-5 流水线

4. 通信度量

并发行为(或进程)之间的通信通常表示为消息通过抽象通道(channel)传送。每个通道在两个行为之间传输固定大小的消息——信息由一个称为产生者(producer)的行为产生,由另一个称为消费者(consumer)的行为接收。在其生存期内,产生者行为将发起若干次通道数据传输。数据传输速率,或称比特率(bitrate),用于度量通信通道传输的快慢。例如,在图 7-4b 中,进程 *P* 通过通道 *C* 接收从进程 *Q* 发来的数据,其最大速率为 10 Mb/s。

244

对于每个通信通道,可定义两种数据传输速率:平均速率与最高速率。通道 *C* 的平均速率(average rate) $avgrate(C)$ 定义为两个进行通信的行为在整个生存期中数据传输的速率。平均数据传输速率可通过用发送数据的行为的执行时间去除通道传输的总位数而得到。举例说明,图 7-6 显示出 8 位消息经由通道 *C* 的传输。每个消息占用 100 ns 的通道时间。由于该通道在 1000 ns 时间内发送 56 位的数据,因此通道 *C* 的平均速率计算如下:

$$avgrate(C) = \frac{56 \text{ bits}}{1000 \text{ ns}} = 56 \text{ Mb/s}$$

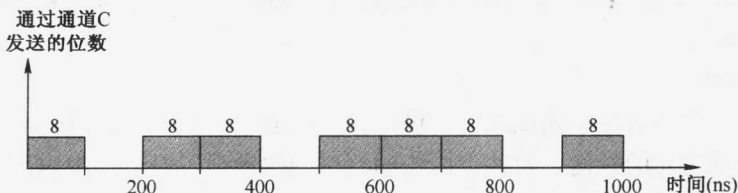


图 7-6 通道平均速率和最高速率的计算

通道 *C* 的最大速率(peak rate) $peakrate(C)$ 定义为数据以单个消息在通道上传送的速率。最大速率可通过用消息传送所需时间去除该消息所携带的位数来得到。对于图 7-6 中的通道 *C*,每个消息传送需要 100 ns。由于每条消息的大小是 8 位,故其最大速率为

$$peakrate(C) = \frac{8 \text{ bits}}{100 \text{ ns}} = 80 \text{ Mb/s}$$

在系统级对数据传输速率进行评估具有本质上的重要性,原因有以下方面:首先,由于每个通道都由总线来实现,而总线包括一组导线和控制导线内数据传输的协议,因此通信通道的数据传输速率将直接影响总线宽度。因此,要实现高平均速率和最高速率的通道就需要一个比较宽的总线。

245

其次,通道速率直接影响到经由通道通信的两个行为的执行时间。比较慢的数据传输率会使得消费者行为和生生产者行为在通道通信上花费更长的时间,从而导致更长的行为执行时间。

最后,当行为被划分到各个芯片上,通道被划分到总线上时,通信速率扮演了一个重要角色。经由通道以高比特率进行通信的行为更有可能被分配到相同的芯片中,使得片外访问时延最小化。而且,具有很高比特率的通道不太可能分到同一个总线中,这样就使得当若干个行为试图同时向同一个总线传输数据时,使性能退化达到最小化。

5. 事件间时序

设计者可能经常需要为设计描述中的一对事件规定时序关系。这种时序关系表示两个事件发生之间的时间间隔。例如,在图 7-4c 中,设计者规定:信号 *OUT* 的有效时刻不能早于信

号 IN 有效之后 50 ns。事件间时序表示了事件之间的约束,一般用于时序图中。时序图表示系统各组件间的接口。两事件之间指定的约束可以直接转换为针对计算的约束,而该计算需要在这两个事件发生之间执行。

7.2.4 其他度量

246

现在介绍几个也可应用于系统设计的质量度量。

1. 功率耗散

系统组件的功率耗散是指在 CMOS 电路进行开关转换过程中负载电容充放电引起的能量耗散。功率耗散取决于系统的时钟频率和在每个时钟周期中实际上处于活跃(即发生值的变化)的门数量。时钟频率越高,组件中的活跃门数量越多,组件的功率耗散就越大。

对功率耗散进行评估有几个方面的益处。在电池驱动的系统,大功率需求要求用大电池,这样就导致产品成本和重量的明显提升。由于过高的功率耗散会造成组件失效,所以,设计中可能会耗散大量功率的部分一般不把它们分配到一个封装中,以避免在后续阶段出现组件失效。最后,若试图通过提高时钟频率来提高设计性能,就必须考虑到小的时钟周期所导致的功率耗散增加。

2. 可测性设计

可测性设计产生具有最小测试成本的设计。测试成本实质上是内置的测试硬件和设计实现后测试成本两者之间的权衡。组件中内置的测试硬件增强了对设计中内部状态的可控性和可观性,因而降低了组件实现后的测试时间。可控性(controllability)定义的是为了达到测试目的而将内部逻辑初始化为特定的控制状态的能力。可观性(observability)指的是从外部对设计的内部状态进行观测的能力。

247

可测性影响许多系统级的参数。有可能需要增添额外的测试引脚以提高可控性和可观性,并导致封装成本和封装装配时间的增加。此外,由于测试电路的增加和更多的引脚驱动,额外引脚将导致面积与功率耗散的增加。而且,由于内置测试硬件,电路复杂性增加,成品率将会大幅度下降。因此,可测性设计在降低测试开销的同时,也增加了生产成本。

3. 设计时间

设计时间定义为从功能描述到获得设计实现所需的时间。设计时间取决于所采取的设计方法。例如,给定系统的英文描述文档,自动化设计方法所采用的抽象级别对设计时间有显著的影响。用高级语言表示系统,并采用高层次综合、逻辑综合及物理设计工具,要比用人工从功能描述转换到晶体管原理图,所需要的时间要少得多。减少设计时间的方法还有:采用更多的可编程组件和现货供应的组件,使用较少的定制组件,因为定制组件需要大量的设计和测试开销。

4. 上市时间

上市时间也许是推动系统设计的唯一最重要的因素,尤其是因为任何设计的利润潜力都与其进入终端用户的早晚有直接的正比例关系。上市时间可定义为从设计概念化到产品实际交付到消费者手中所经历的总时间。除了设计时间,它还包括对市场进行调研,对用户需求进行描述定义、制造、测试、销售,以及为了使设计可用而进行的支撑软硬件开发等所需的时间。

5. 制造成本

制造成本度量的是与设计最终实现相关的财政费用问题。制造成本包括人力、原材料、制

造、封装、测试、维护设备等所需的成本。

248

7.3 硬件评估

7.3.1 硬件评估模型

在本节中我们将讨论对于硬件实现进行质量度量评估的一般设计模型。

给定要进行评估的设计描述,假定相应的设计用 FSMD 实现。如在第 2 章所述,FSMD 体系结构包含一个控制单元和一个数据通路,如图 7-7 所示。

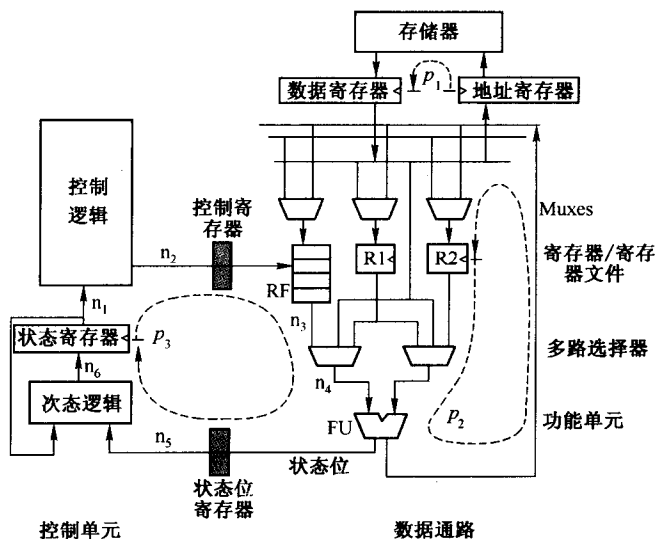


图 7-7 用于评估的控制单元/数据通路模型

数据通路包含寄存器、寄存器文件、功能单元及多路选择器。一个典型的数据通路有一个两级的多路选择器结构,如图 7-7 所示。在数据通路上的典型操作是从寄存器或寄存器文件中读取操作数,在功能单元中计算结果,并最终将结果写入到目的寄存器中。所有存储器存取都是经由寄存器通过读取和存储操作实现,即在数据通路使用某数据之前将其读取到寄存器中,并在计算结果写到存储器之前将其存储到寄存器中。

控制单元包括状态寄存器(state register),用于驱动数据通路组件控制线的控制逻辑(control logic),以及用于计算次态(将存储于状态寄存器中)的次态逻辑(next-state logic)。来自数据通路的状态线携带了次态逻辑所需要的比较操作的结果。

为了弄清控制逻辑和数据通路的时延是如何影响系统时钟周期的,请见图 7-7 中设计模型的各个路径。该设计中每条路径定义了一个数据流,始于某一寄存器,终于另一寄存器,且必须在一个时钟周期内完成。路径 p_1 是关于读存储器的,它始于地址寄存器 AR,途经存储器并终止于数据寄存器 DR,DR 就是从存储器取来的数据要写入的寄存器。该路径的时延包括读地址寄存器的时延,访问存储器的时间,数据寄存器的建立时间以及这些组件间的互联网的时延;设计中另一个寄存器至寄存器的路径 p_2 ,首先从寄存器组读入数据,通过多路选择器发送至功能单元,并将计算结果通过另一个多路选择器返回到寄存器组;另外一个路径 p_3 ,始于状态寄存器,途经控制逻辑、寄存器文件、多路选择器、功能单元,最后经过次态逻辑返回至

249

状态寄存器。

为了完成设计中每条路径上的数据传输,时钟周期 clk 必须大于设计中最长路径的时延。比如途经控制逻辑的路径 p_3 具有最大的时延。因此,最小的时钟周期必须等于或大于路径 p_3 上的所有组件和线网的时延总和。这样,如果暂时忽略控制和状态位寄存器(status register),我们需要遵守如下条件:

250

$$clk \geq delay(SR) + delay(CL) + delay(RF) + delay(Mux) + delay(FU) + delay(NS) + setup(SR) + \sum_{1 \leq i \leq 6} delay(n_i) \quad (7-6)$$

其中:

$delay(SR)$ 表示读状态寄存器的时延;

$delay(CL)$ 表示控制逻辑的时延;

$delay(RF)$ 表示读寄存器文件的时延;

$delay(Mux)$ 表示多路选择器的时延;

$delay(FU)$ 表示功能单元的时延;

$delay(NS)$ 表示次态逻辑的时延;

$setup(SR)$ 表示状态寄存器的建立时间;

$delay(n_i)$ 表示与线网 n_i 相关的时延。

很显然,如上式计算的时钟周期可能过长,从而导致设计性能的显著降低。时钟周期可采用控制流水线(control pipelining)来减小[GDWL91, RG93],即在数据通路和控制单元之间的适当点处插入寄存器。图 7-7 显示出如何插入控制和状态位寄存器以减少时钟周期。例如,如果在数据通路与控制单元之间插入了这两个寄存器,时钟周期就成为三条路径中的最大时延:一条从状态寄存器到控制寄存器,另一条从控制寄存器途经数据通路到状态位寄存器,第三条从状态位寄存器到状态寄存器。

在介绍了系统硬件实现的设计模型之后,我们将在下面几节中讲述质量度量的计算方法,包括系统时钟周期、控制步数、执行时间、通信速率、设计面积、以及引脚等。

7.3.2 时钟周期评估

在 7.2.3 节中,我们介绍过对时钟周期的选择可以影响执行时间和设计所需的资源数。因此,在执行系统设计之前对时钟周期进行评估至关重要。

251

在大多数综合工具中[WC91, PK89a, PKG86, BM89, MK90],设计者在综合之前必须对时钟周期做出规定。或者明确规定时钟周期,或者将组件的时延表示成时钟周期的倍数。当进行的设计是一个大型系统的一个组成部分时,可由设计者规定时钟周期。在这种情况下,系统中一些标准组件的时钟周期已知,并可用于设计的其余部分。

当设计者没有规定时钟周期时,就需要进行评估。在本节,我们将提出评估系统时钟周期的方法。

1. 最大操作时延法

一些综合工具[PPM86, PP85, JMP88]将时钟周期等同于设计中最长的操作时延。假设 $delay(t_i)$ 表示为实现 t_i 类型操作的功能单元所需要的时延。那么,用最大操作时延法进行评估,时钟周期 $clk(MOD)$ 计算如下:

$$clk(MOD) = \text{Max}_{\text{所有 } t_i} (delay(t_i)) \quad (7-7)$$

例如,考虑二阶微分方程的例子[PKG86],它包括加法、减法和乘法操作。假设:

$$\text{delay}(+) = 49 \text{ ns}, \quad \text{delay}(-) = 56 \text{ ns}, \quad \text{delay}(\times) = 163 \text{ ns}$$

由式 7-7,可以将时钟周期评估为以上三种功能单元中的最大时延,即 $\text{clk}(\text{MOD}) = 163 \text{ ns}$ 。

最大操作时延法的优点是对时钟周期的评估速度非常快而且易于实现。然而,采用最大操作时延作为时钟将导致对较快功能单元的利用不足。其原因是,当存在较慢功能单元,比如具有大时延乘法器的情况下,时钟周期会至少等于乘法器的时延,而实现其他操作的较快的功能单元将在时钟周期的很大部分时间内处于空闲状态。因此,采用最大操作时延法,就希望设计具有较长的执行时间。

252

2. 时钟松弛

为了改进设计的性能,我们需要使功能单元的空闲时间最小化。功能单元的时钟松弛(clock slack)就表示该功能单元在一个时钟周期内空闲的那部分时间。松弛(slack)定义为功能单元的时延与下一个更高倍时钟周期的差。与前面一样,我们定义 $\text{delay}(t_i)$ 为实现 t_i 类型操作的功能单元所需要的时延。对于一个给定的时钟周期 clk 和操作类型 t_i ,相应的功能单元的松弛 $\text{slack}(\text{clk}, t_i)$ 由下式进行计算:

$$\text{slack}(\text{clk}, t_i) = (\lceil \text{delay}(t_i) \div \text{clk} \rceil \times \text{clk}) - \text{delay}(t_i) \quad (7-8)$$

我们一般都假定,对应于某功能单元,比较小的松弛将会导致对该功能单元更充分地使用,以及在相同数量资源的情况下达到更短的执行时间。

图 7-8 描绘了针对微分方程例子的不同操作所对应的松弛。假设用最大操作时延法来确定时钟周期。由于乘法器有最大时延,故时钟周期 $\text{clk}(\text{MOD}) = 163 \text{ ns}$ 。采用式 7-8,我们可以算出以下松弛:

$$\text{slack}(163, \times) = (\lceil 163 \div 163 \rceil \times 163) - 163 = 0 \text{ ns}$$

$$\text{slack}(163, -) = (\lceil 56 \div 163 \rceil \times 163) - 56 = 107 \text{ ns}$$

$$\text{slack}(163, +) = (\lceil 49 \div 163 \rceil \times 163) - 49 = 114 \text{ ns}$$

图 7-8 中浅阴影区域表示用于实现设计的加法器、减法和乘法器的时延。对于 163 ns 的时钟周期,加法器和减法器都有比较大的松弛,如图中的黑色区域。若时钟周期比较长,并且在时钟周期的很大一部分时间内某些功能单元是空闲的,则必定会导致较长的运行时间,正如在该设计中采用 163 ns 时钟的情况。

253

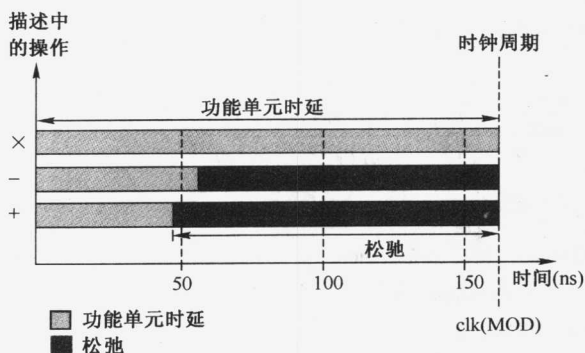


图 7-8 163 ns 时钟周期下的功能单元的松弛

3. 最小松弛法

最小松弛法的思想是[NG92]:在假定比较小的松弛能够促进功能单元的充分利用,并依此降低行为执行时间的前提下,使每个时钟周期的松弛最小化。

式7-8给出了功能单元松弛的计算式。对于给定的时钟周期 clk , 平均松弛 (average slack) $ave_slack(clk)$ 定义为时钟周期的一部分, 在该部分中的每个单元在平均状况下都处于空闲。如果用 $occur(t_i)$ 表示行为中 t_i 类型操作发生的数量, T 表示操作类型的数量 (如加、减、乘), 那么平均松弛如下式计算:

$$ave_slack(clk) = \frac{\sum_{i=1}^T (occur(t_i) \times slack(clk, t_i))}{\sum_{i=1}^T occur(t_i)} \quad (7-9)$$

时钟利用率 (clock utilization) 定义为设计中所有功能单元进行的有效计算在一个时钟周期内所占的百分比, 也就是说:

$$utilization(clk) = 1 - \frac{ave_slack(clk)}{clk} \quad (7-10)$$

最小松弛法检验的是潜在的时钟周期范围, 计算的是每个时钟周期的利用率。对于某个行为 B , 使其时钟利用率最大的时钟周期就可作为最小松弛时钟。

算法 7.3.1: 松弛最小化

```

/* 确定每种操作类型发生的次数 */
ComputeRange(  $T$ ,  $clkmax$ ,  $clkmin$  )
for each  $t_i \in T$  loop
     $occur(t_i) = \text{FindOccurrences}(B, t_i)$ 
end loop

/* 计算每种可能时钟的利用率 */
 $max\_utilization = 0$ 
for  $clkmin \leq clk \leq clkmax$  loop
    for each  $t_i \in T$  loop
         $slack(clk, t_i) = (\lceil delay(t_i) \div clk \rceil \times clk) - delay(t_i)$ 
    end loop
     $ave\_slack(clk) = \frac{\sum_{i=1}^T [occur(t_i) \times slack(clk, t_i)]}{\sum_{i=1}^T occur(t_i)}$ 
     $utilization(clk) = 1 - \frac{ave\_slack(clk)}{clk}$ 
    if  $utilization(clk) > max\_utilization$  then
         $max\_utilization = utilization(clk)$ 
         $max\_utilization\_clk = clk$ 
    end if
end loop

```

```
return(max_utilization_clk)
```

算法 7.3.1 描述了如何利用最小松弛法进行时钟评估。不同的操作类型数用 T 表示。过程 *ComputerRange* 用于确定时钟周期的范围,这些时钟周期将会由算法来检验,检验方法是考察用于实现行为的 T 类操作中的每个操作所需功能单元的时延。时钟周期的范围用 $(clkmin, clkmax)$ 表示。所有功能单元的最大时延用 $clkmax$ 表示。设计库通常规定了双稳态电路的最大时钟输入频率,使得能在逻辑级保持稳定的转换。该频率用于确定 $clkmin$ 的值。在没有规定最大时钟频率的情况下, $clkmin$ 近似为 $delay(t_i)$ 的最小值。

255

函数 *FindOccurrences* 对行为 B 的行为描述进行检查,并返回其中 t_i 类型操作的发生次数。实现了 t_i 类型操作的功能单元的时延表示为 $delay(t_i)$ 。

对于每个处于范围 $(clkmin, clkmax)$ 内的时钟周期 clk ,最小松弛算法利用式 7-8 对每个实现了行为中某个操作的功能单元的松弛进行计算。这样,平均松弛就可由式 7-9 算出。最后,当前时钟周期 clk 的利用率可由式 7-10 算出。由算法 7.3.1 得出的最大时钟利用率保存在 $max_utilization$ 中,相应的时钟周期保存在 $max_utilization_clk$ 中。使时钟利用率最大化的时钟周期值作为最小松弛时钟返回。

操 作	$occur(t_i)$	$delay(t_i)$
加	2	48 ns
减	2	56 ns
乘	6	163 ns

图 7-9 微分方程例子中操作发生次数和操作时延

我们通过将该算法应用到二阶微分方程例子[PKG86]来进行说明。每个操作发生的次数和用于实现这些操作的功能单元的时延如图 7-9 所示。其中用到的组件来源于 VDP100 数据通路库[VTI88]。

256

在图 7-10a 中,功能单元的时延图示为沿 X 轴的浅阴影区域长度。行为操作出现的次数表示为沿 Y 轴的阴影区域的高度。

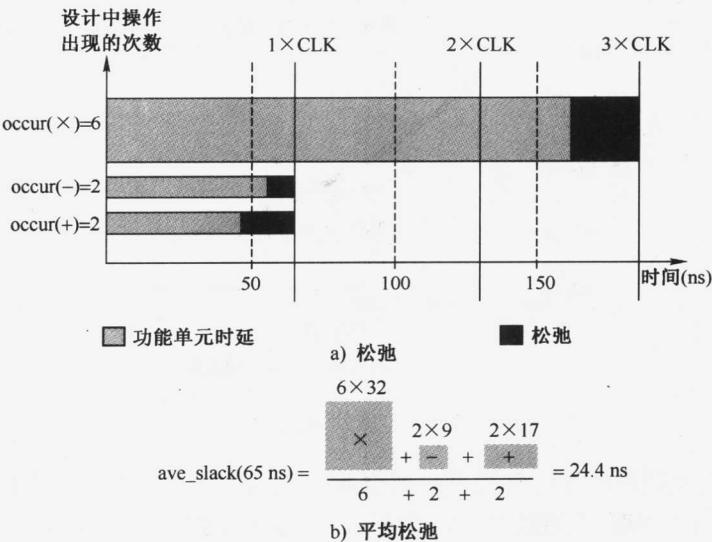


图 7-10 计算时钟周期为 65 ns 的情况

由于 VDP100 库规定时钟寄存器的最大频率为 75 MHz,所以 $clkmin = 1/75\text{ MHz} = 14\text{ ns}$ 。

最大功能单元时延为 $clkmax = delay(\times) = 163 \text{ ns}$ 。

257

我们可以计算出时钟周期为 65 ns 的时钟利用率。利用式 7-8, 我们可以得出:

$$slack(65, \times) = (3 \times 65) - 163 = 32 \text{ ns}$$

$$slack(65, -) = (1 \times 65) - 56 = 9 \text{ ns}$$

$$slack(65, +) = (1 \times 65) - 48 = 17 \text{ ns}$$

在图 7-10a 中的黑色阴影区域表示的是每种操作类型的松弛。针对 65 ns 时钟周期的平均松弛可由式 7-9 算出, 并在图 7-10b 中显示出来。平均松弛为 24.4 ns 。

最后, 利用式 7-10 我们可以得出 65 ns 时钟周期的时钟利用率:

$$utilization(65 \text{ ns}) = 1 - \frac{24.4}{65} = 0.62 = 62\%$$

针对从 14 ns 到 163 ns 的所有时钟周期值, 重复进行时钟利用率计算。在 56 ns 时钟周期处, 达到了最大时钟利用率 92% , 如图 7-11a 所示。该值被选做最小松弛时钟 $clk(SM)$ 。一个有趣的发现是, 最大操作时延法的所选择的 163 ns 时钟周期仅有 73% 的利用率。

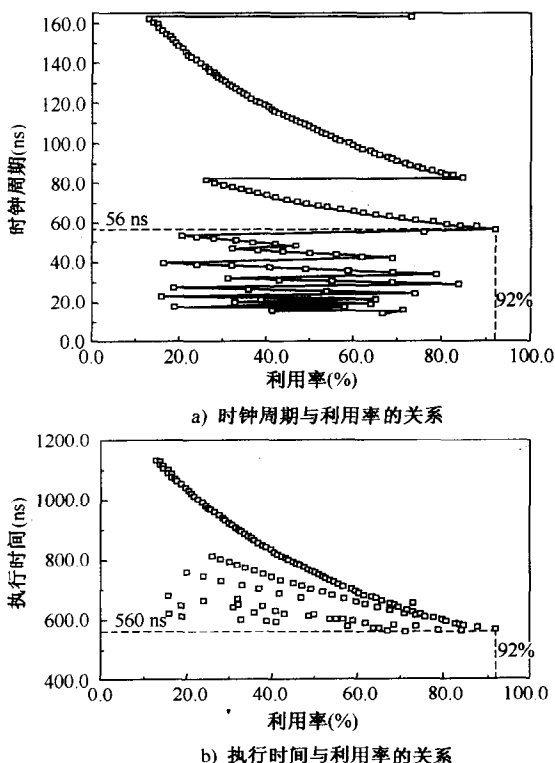


图 7-11 微分方程例子

为表明具有最大利用率的时钟周期确实会导致比较快的执行速度, 对于每个从 $clkmin$ 到 $clkmax$ 范围内的时钟周期, 可通过对行为描述进行调度并用控制步数乘以时钟周期的方法来确定实际执行时间。图 7-11b 描绘出微分方程例子的执行时间相对于时钟利用率的变化。图中显示出, 最小松弛时钟 56 ns 会得到 560 ns 的执行时间, 并具有 92% 的利用率。只有一个时钟周期 (15 ns), 其利用率为 71% , 可以导致一个达到边际的快速设计, 执行时间可达 555 ns 。

实验表明,采用最小松弛法进行评估所实现的设计,要比用最大操作时延法所实现的设计,性能上要好 32% [NG92]。不管在设计实现过程中资源如何分配,执行时间还是减少了。

在文献[GSV92]中对最小松弛算法做出了两点修正。首先,将多周期操作中的中间时钟周期引入到时钟利用率计算中,与式 7-10 计算出的利用率相比,该方法得出了关于执行时间和时钟利用率倒数之间更好的相互关系;其次,不是对从 $clkmin$ 到 $clkmax$ 整个范围内的时钟周期进行检验,而仅仅对是功能单元时延因子的时钟周期进行检验,因而大大降低了算法的计算时间。

7.3.3 控制步评估

行为执行所需的控制步数可通过若干种方法评估。在本部分中,我们首先讲述两种技术——运算符-使用(operator-use)方法和调度算法,用来评估定义成直线型代码的行为的控制步数。然后,我们针对包含有分支和迭代控制结构的行为提出一种对控制步数的评估方法。

1. 运算符-使用方法

运算符-使用方法在给定实现行为的所需资源的情况下,评估执行该行为的控制步数。该方法将行为中的所有语句划分成一组结点,使得一个结点内的所有语句可并发执行。

设 T 表示行为 B 中不同类型操作的数目, $num(t_i)$ 和 $clocks(t_i)$ 分别表示为实现 t_i 类型操作时,可以采用的功能单元的数目和时延(用时钟周期数表示)。那么,在任意一个结点中,如果 t_i 类型操作出现(或使用) $occur(t_i)$ 次,则至少需要 $\left\lceil \frac{occur(t_i)}{num(t_i)} \times clocks(t_i) \right\rceil$ 个控制步来执行 t_i 类型操作。任一结点 n_j 所需的控制步数 $csteps(n_j)$ 等于在结点中运行任何类型操作所需的最大控制步数,即:

$$csteps(n_j) = \max_{t_i \in T} \left\lceil \left[\frac{occur(t_i)}{num(t_i)} \right] \times clocks(t_i) \right\rceil \quad (7-11)$$

一旦每个结点的控制步数已定,则行为 B 中所需的总控制步数就可如下确定:

$$csteps(B) = \sum_{n_j \in N} csteps(n_j) \quad (7-12)$$

其中 N 为 B 中的结点数。

算法 7.3.2: 操作符-使用方法

```
CreateNodes( $B, N$ )
 $csteps(B) = 0$ 
for each node  $n_j \in N$  loop
   $csteps(n_j) = 0$ 
  /* 对于在当前结点中的所有操作类型  $T$  */
  for each operation type  $t_i \in T$  loop
    /* 计算结点中  $t_i$  操作类型的控制步 */
     $csteps(n_j, t_i) = \left\lceil \left[ \frac{occur(t_i)}{num(t_i)} \right] \times clocks(t_i) \right\rceil$ 
    /* 计算结点中的最大控制步数 */
    if  $csteps(n_j, t_i) > csteps(n_j)$  then
       $csteps(n_j) = csteps(n_j, t_i)$ 
```

```

end if
end loop
csteps(B) = csteps(B) + csteps(nj)
end loop
return csteps(B)

```

算法 7.3.2 显示出操作符-使用法如何计算行为 B 的总控制步数。过程 CreateNodes 将行为 B 中的语句划分成为 N 个结点的集合。将行为中的语句合并到结点中所采取的方式是,保持语句间的依赖关系,并使结点总数最少。如果语句 S_2 依赖于语句 S_1 ,则 S_2 所分配到的结点应是 S_1 所分配到的结点的后继。操作符-使用法的这种将语句聚合成结点的过程与时钟周期和为该设计配置的资源数量无关。

对于每个结点,算法 7.3.2 计算出完成 T 个不同操作类型中的每个操作所需控制步数。变量 $csteps(n_j, t_i)$ 表示在结点 n_j 中为计算 t_i 类型操作所需的控制步数。式 7-11 计算得出的变量 $csteps(n_j)$ 表示为执行结点 n_j 中的所有操作所需的控制步数。将每个结点所确定的控制步数相加,如式 7-12 所示,就确定了整个行为的总控制步数。

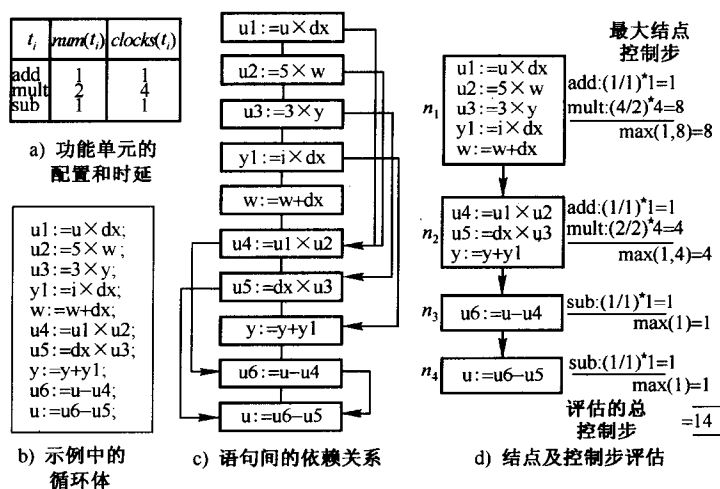


图 7-12 应用于微分方程例子的操作符-使用法

以微分方程为例说明操作符-使用法。图 7-12a 示出的是加法器、减法和乘法器的资源配置和时钟周期时延的情况。行为的主体示于图 7-12b。单个语句之间的依赖关系在图 7-12c 中用箭头表示。在图 7-12d 中,语句被合并成为四个结点 n_1 、 n_2 、 n_3 、 n_4 。请注意在将语句合并成结点的过程中,没有违反原始行为中的依赖关系。换句话说,如果两个状态互相依赖,那么为了保持这种依赖性,它们将被分配到两个不同的结点。

图 7-12d 显示出采用式 7-11 对每个结点计算控制步数的情况。例如,结点 n_1 包括一个加法操作和四个乘法操作,即 $occur(+)=1$ 和 $occur(\times)=4$ 。由于仅配置了两个乘法器且每次乘法需要四个时钟周期,则这四个乘法操作将需要至少八个控制步。类似地,该结点中的单个加法操作需要一个控制步来执行。利用式 7-11,我们可以估计出结点 n_1 至少需要八个控制步来执行其操作。利用式 7-12 将行为中所有结点所需的控制步数加起来,就可以估计出需

要 14 个控制步来实现微分方程的行为。

操作符-使用法还可以扩展使之能结合进流水线功能单元和包含存储器访问。例如,请考虑一个具有固定访问时间的多端口存储器。如果我们将存储器看做一个功能单元,那么结点对存储器的访问次数就对应于操作发生的次数,访问存储器的有效端口数就等于为每个操作所分配的功能单元数,并且内存访问时间与功能单元的时延相同。因此,现在就可以直接利用式 7-11 来确定结点在有存储器访问情况下的控制步数。

操作符-使用法可以对行为所需的控制步数提供相当快速的评估。如果行为中含有 n 个操作,则该方法的计算复杂度为 $O(n)$ 。但是,它有出错的可能性,其原因是该方法工作于语句级的粒度上,忽略了语句内部各操作之间的依赖关系。比如,如果有两个具有一个时钟周期时延的加法器,则采用该方法可以得出结论:语句 $A: = B + C + D$ 可以在一个控制步完成。但是,事实上需要两个控制步:一个用来计算部分和“ $B + C$ ”,另一个用来将 D 加入部分和。总之,如果在系统描述中的每个语句限定为一个操作,操作符-使用法就会产生更准确的评估。

2. 调度

调度技术可应用到行为中以确定控制步数。给定资源约束,以总控制步数最小化为目标来调度行为中的操作。在本部分中,我们将讲述这样一种具有资源约束的调度算法——列表调度。有关调度算法的更全面的讨论请见[GDWL91]。

对于行为中每种类型的操作,列表调度算法都保持一个优先表,通过它将操作分配到控制步中。在任何一个阶段, t_i 类型操作的优先表包含了一组操作,该组操作的前驱操作均已被调度。为执行某操作,需要另一些操作的结果,“前驱操作”指的就是这另一些的操作。因此,某操作的所有前驱操作必须先于该操作之前执行。

算法 7.3.3 描述了列表调度算法。和前面一样,设 T 表示行为中操作的类型数量, $cstep$ 表示操作正在被调度的当前控制步。 t_i 类型操作的优先表用 $plist(t_i)$ 表示。给定一个表,函数 First 返回表中第一个元素,函数 Tail 返回的是去掉首个元素后的该表。过程 CreatePriorityList 对每种操作类型的优先表进行初始化,使它们仅包含无前驱的操作,而过程 UpdatePriorityList 则扫描那些未被调度的操作,选择那些其前驱均已被调度的操作,并将所选择的操作加入到适当的优先表中。过程 UpdateSchedule 通过将指定操作分配到当前控制步中以达到对调度 S 进行修改的目的。

算法中每次迭代都对应于对一组操作的选择,这组操作将被分配到当前控制步中。如果 $num(t_i)$ 个功能单元被分配给 t_i 类型操作,那么,在每个控制步,算法会从相应优先表中取出前 $num(t_i)$ 个操作分配到当前控制步中。每个被分配到当前控制步中的操作都会从优先表中删除,调度也将被更新以反映该次分配的效果。在所有可能的操作都被分配到当前控制步后,优先表被更新。该算法反复执行直至给定行为中的所有操作都被调度。行为所需控制步的数量即为算法终止时变量 $cstep$ 的值。

算法 7.3.3: 列表调度

```
CreatePriorityList(  $V, plist(t_1), plist(t_2), \dots$  )
 $cstep = 0$ 
while (( $plist(t_1) \neq \phi$ ) or ( $plist(t_2) \neq \phi$ ) or ...) loop
```

262

263

264

```
cstep = cstep + 1
for i in 1 to T loop
  for j in 1 to num(ti) loop
    if plist(ti) ≠ ∅ then
      UpdateSchedule(S, First(plist(ti)), cstep)
      plist(ti) = Tail(plist(ti))
    end if
  end loop
end loop
UpdatePriorityList(V, plist(t1), plist(t2), ...)
```

end loop

列表调度中的一个重要问题是对合适的优先权函数的选择,该函数用于对优先表中的操作进行排序。这种优先权函数是操作的机动性,定义为某操作可能被分配到的控制步数,并且这种分配不会时延整个行为的完成时间。一个操作的机动性越低,其调度优先级就越高,原因是尽早调度可以使该操作更早地选择控制步。

265

虽然调度算法使设计者获得执行行为所需要的准确的控制步数,但计算代价很高。例如,列表调度的复杂度是 $O(n^2)$,其中 n 表示行为中的操作数。

图 7-13 将采用操作符-使用法得出的评估值与采用基于机动性的列表调度算法 [Lis92] 得到的实际控制步数进行了比较。这些比较是在高层综合中的一些基准实例上,比如椭圆滤波器 [KWK85]、线相 B 样条内插滤波器 [PF89]、微分方程 [PKG86] 以及 AR 格滤波器 [JMP88] 进行的。采用操作符-使用法的平均误差仅为 13%。

设计实例	操作符-使用法	列表调度
椭圆滤波器	22	19
线相 B 样条内插滤波器	6	6
微分方程	14	13
AR 格滤波器	14	11

图 7-13 采用操作符-使用法和列表调度法对控制步数进行评估的结果比较

3. 含有迭代与分支结构的行为

操作符-使用法和调度都可用来评估直线代码所描述行为的控制步数需求。如果一个行为具有分支和迭代结构,则先将它转化为一组基本块。一个基本块(basic block) [ASU88] 是指一个连续的 HDL 语句序列,其中控制流从该语句序列的开头流入,从末尾流出,并且除了在结尾外,不能在中间停止,也不能出现可能的分支。每个基本块由一组顺序赋值语句组成,有时称之为直线代码。

266

在每个基本块中的控制步数可采用以上任意方法来确定。整个行为的控制步数则取决于在互斥分支上的操作是否共享相同的控制步。

共享控制步(shared control step):如果控制步在行为中的各分支间是共享的,则该行为的控制步总数就等于沿最长路径穿越控制流图所需的控制步数。对互斥操作共享控制步会导致控制单元中状态数的减少。但是,这种实现需要有状态位寄存器,它的值用来确定在任一给定

状态,这些互斥操作中的哪一个将被执行。

独立控制步(separate control step):如果在行为中,不同分支的操作被分配给单独的控制步,则行为中控制步总数就等于每个基本块所需要的控制步数之和。这种实现将会使控制单元具有更多的状态。由于每个控制步能唯一辨别出即将在该控制步中执行的操作,所以不需要状态位寄存器。

举个例子,我们来看行为 B 的控制流(control flow),其中包括 o_1 到 o_8 共 8 个操作,如图 7-14a 所示。行为中的操作被划分为四个基本块, B_1 到 B_4 , 在图中用虚线框指出。假设由基本块 B_2 和 B_3 所表示的条件分支中,操作是互斥的。采用共享控制步的调度如图 7-14b 所示。在该调度中,尽管操作 o_3 和 o_6 在不同的分支中,但被分配到相同的控制步骤 s_3 中。在这种情况下,整个行为总共需要 6 个控制步。另一方面,采用独立控制步的调度则需要 8 个控制步骤来实现该行为,如图 7-14c 所示。

267

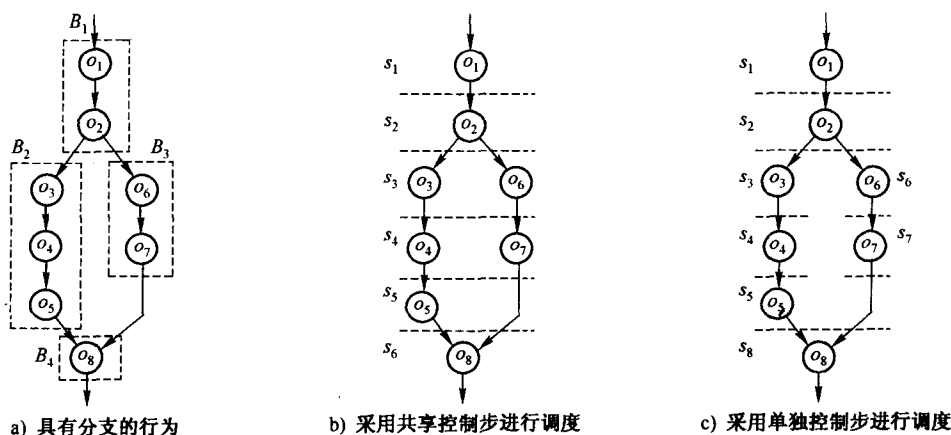


图 7-14 控制步评估

7.3.4 执行时间评估

我们现在来说明针对一个完整的功能描述中行为执行时间的评估方法。

如果一个行为是用一系列直线代码描述的,则该行为自始至终的执行时间取决于该行为被调度的控制步数。设 $csteps(B)$ 表示对行为 B 进行调度的控制步数的评估, clk 表示为实现该设计所选定的时钟周期。那么,该行为的执行时间 $exectime(B)$ 计算如下:

$$exectime(B) = csteps(B) \times clk \quad (7-13)$$

一般情况下,行为由顺序语句组成,并有分支和循环结构(如 loops、if 和 case 语句)。由于分支可能依赖于输入数据,所以不能直接用式 7-13 来确定这种行为的执行时间。在这种情况下,可采用一种基于概率的流分析技术。

我们首先确定行为中的基本块。由于每个基本块都是由一组顺序的赋值语句组成,我们可以为每个基本块确定执行时间,方法是先如 7.3.3 节所述求出所需控制步数,然后再应用式 7-13 来计算。

为确定整个行为的执行时间,需要确定每个基本块 b_i 的执行时间 $exectime(b_i)$,再用该基本块的执行频率(execution frequency)加权。一个基本块 b_i 的执行频率 $freq(b_i)$ 定义为该基本块所属行为在一次执行过程中,该基本块被执行的平均次数。一旦确定了每个基本块的执

268

行频率,则行为的总共执行时间 $exectime(B)$ 就可用下式评估:

$$exectime(B) = \sum_{b_i \in B} exectime(b_i) \times freq(b_i) \quad (7-14)$$

现在,我们提出一种方法以确定行为中每个基本块的执行频率。

基于概率的流分析

为了计算每个基本块的执行频率,我们必须首先为行为中的基本块创建等价的控制流图模型。设 $G=(V,E)$ 为一个图,其中 V 表示顶点 v_i 的集合, E 表示连接顶点 v_i 到 v_j 的有向边 e_{ij} 的集合。

对于行为描述中的每个基本块 b_i ,在图 G 中都对应一个顶点 v_i 。条件语句(if、case 或 loop)的目标块是指对条件求值后,控制可能被传递到的任何基本块。对于任意一个基本块 b_j ,如果它是紧随基本块 b_i 之后的条件语句的目标块,则在 G 中有一条从顶点 v_i 指向 v_j 有向边 e_{ij} 。

在得到了行为的控制流图后,我们就可以采用基于分支概率的流分析法确定每个结点的执行频率。

269

分支概率是在对某分支语句的条件求值后,对某分支执行频率的一个度量。分支概率可通过若干种途径来确定。第一,可以用静态的方式计算概率。对于迭代次数 n 已知的循环语句的情况,则在控制流图中,循环的边被赋予概率 $(n-1)/n$,而退出循环的边被赋予概率 $1/n$ 。对于其他的分支状态如 if 或 case,则可对每个分支赋予相等的概率。第二,设计者可采用在设计描述中加入注解或在运行时进行交互的方式,直接在行为的分支结构中指定概率。第三,概率可动态地获得,方法是在若干种样本数据集上对行为进行模拟,记录下各个分支被执行的频率,并因此得出每个分支的概率。

我们现在可以通过分析控制流图中边的执行概率来确定各结点的执行概率。其过程概述如下:

1) 在给定控制流图中的第一个结点之前加入一个开始结点 S 。由于该结点仅仅在控制流图被执行时才执行 1 次,故将其执行频率 $freq(S)$ 设为 1。

2) 任何一个结点 v_j 的执行频率取决于其所有直接前趋结点的加权执行频率。每个前趋结点 v_i 的加权执行频率是将其执行频率乘以 v_i 和 v_j 之间边的分支概率 $prob(e_{ij})$ 。我们首先将图中每个结点的执行频率用方程表示出来。对于控制流图中的任一结点 n_j

$$freq(n_j) = \sum_{\text{所有前驱结点 } n_i} freq(n_i) \times prob(e_{ij}) \quad (7-15)$$

3) 由上一步表示的线性方程组可用高斯消去法或 LU 因子分解等方法得到各结点的执行频率。这些也就是行为中相应基本块的执行频率。

我们以图 7-15a 所示的 VHDL 行为为例来说明基于概率的流分析法。VHDL 语句的基本模块如图 7-15b 所示。对于行为中的每个基本块,在图 7-15c 的控制流图中都有一个顶点。如果控制可在相应的基本块中传递的话,则在这两顶点间存在一条边。例如,如果条件 $D > A$ 为真,控制就可以从基本块 B_2 到 B_3 传递。这样,在顶点 v_2 和 v_3 之间存在一条边 e_{23} 。此外,还增加了一个虚起始结点 S 。

270

在图 7-15c 中还标识出在结点 v_2 和 v_5 处的分支概率。我们现在可以用方程来表示出控制流图中各结点的执行频率。例如,结点 v_2 有两个前趋结点 v_1 和 v_5 。结点 v_1 每执行一次

的时间为结点 v_2 执行一次的时间加上结点 v_5 执行时间的 0.9 倍。因此

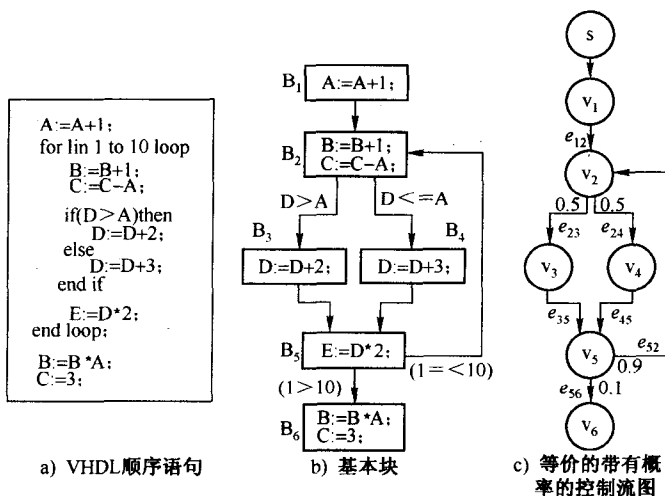


图 7-15 构造行为的控制流程图

$$freq(v_2) = 1 \times freq(v_1) + 0.9 \times freq(v_5)$$

继续用类似的方式来看图 7-15c 中的其他结点,我们得到如下方程组:

$$freq(S) = 1.0$$

$$freq(v_1) = 1.0 \times freq(S)$$

$$freq(v_2) = 1.0 \times freq(v_1) + 0.9 \times freq(v_5)$$

$$freq(v_3) = 0.5 \times freq(v_2)$$

$$freq(v_4) = 0.5 \times freq(v_2)$$

$$freq(v_5) = 1.0 \times freq(v_3) + 1.0 \times freq(v_4)$$

$$freq(v_6) = 0.1 \times freq(v_5)$$

通过对得到的线性方程组求解,就可求出图中所有结点的执行频率:

$$freq(v_1) = 1.0 \quad freq(v_2) = 10.0$$

$$freq(v_3) = 5.0 \quad freq(v_4) = 5.0$$

$$freq(v_5) = 10.0 \quad freq(v_6) = 1.0$$

一旦确定了每个基本块的执行频率,即可应用方程 7-14 求出整个行为的执行时间。

在上述方法中,我们是通过每个结点赋予相应基本块的执行时间,并进行流分析来获得整个行为的执行时间。通过对控制流图中每个结点赋予不同类型的信息,基于概率的流分析也可用于确定其他有用的设计特性。比如,如果控制流图中的每个结点表示在相应的基本块中存储器访问的次数,那么流分析将得出在整个行为执行期间存储器访问的平均次数。类似地,如果我们对每个结点赋予某过程在基本块被调用的次数,就可以确定整个行为对该过程的调用次数。最后,通过对每个结点赋予数据在相应基本块中通道上的传输次数,流分析将得出整个行为对通道访问的总次数。

7.3.5 通信速率评估

通信通道或者在行为描述中明确地规定,或者当某个变量被某行为访问时生成(该行为在

272 系统划分过程中被分配到不同于该变量所在的芯片)。并发性之间通信的平均传输速率和最高传输速率在 7.2.3 节中已定义。行为 B 针对通道 C 的平均访问次数 $access(B, C)$ 可采用上一节介绍的流分析方法来计算。

设 $bits(C)$ 表示单个消息中通过通道 C 进行接收或发送的位数。如果一个行为正通过通道访问一个数组变量, 则 $bits(C)$ 也包括了地址的大小。例如, 如果某个行为分别通过通道 chX 和 chY 访问一个 16 位标量 X 和 $32 \text{ 字} \times 16 \text{ 位}$ 的数组变量 Y , 则 $bits(chX)$ 为 16, 而 $bits(chY)$ 为 21(5 位地址和 16 位数据)。

在行为 B 的生存期内, 通过通道 C 发送的总位数如下计算:

$$total_bits(B, C) = access(B, C) \times bits(C) \quad (7-16)$$

任何行为的总执行时间都包括两部分: 计算时间和通信时间。

计算时间(computation time), 即 $comptime(B)$ 定义为行为 B 执行其内部计算所需时间。这些计算表现了行为中诸如赋值、循环、及条件语句的执行。

通信时间(communication time)定义为某行为访问该行为外部数据所需时间。这种通信可表现出通过通道对其他行为中变量的访问。行为 B 通过通道 C 传输数据所需通信时间记为 $commtime(B, C)$ 。

计算时间可采用前一节中讲述的流分析法计算出来。以下我们将给出对于通信速率进行评估的公式。设 $protdelay(C)$ 表示通过通道 C 传输单个消息的时延。为简化表达, 假设行为 B 仅有一个通道 C , 要么接收数据, 要么发送数据, 则该行为的通信时间 $commtime(B, C)$ 计算如下:

273
$$commtime(B, C) = access(B, C) \times protdelay(C) \quad (7-17)$$

通道 C 的平均数据传输速率 $avgrate(C)$ 可如下确定:

$$avgrate(C) = \frac{total_bits(B, C)}{comptime(B) + commtime(B, C)} \quad (7-18)$$

通道的最高传输速率则为:

$$peakrate(C) = \frac{bits(C)}{protdelay(C)} \quad (7-19)$$

7.3.6 面积评估

对任何设计的大小进行评估的首要任务是确定实现给定行为所需组件的数量和类型。一旦所需组件得到确定, 我们就可以获得多种技术条件下的尺寸评估。例如, 对于一个 FPGA 的实现, 对设计中采用的组合逻辑块(combination logic block, CLB)总数的评估可以通过将设计中每个组件采用的 CLB 数目相加得到。对于一个门阵列的实现, 设计复杂度可以通过将组件所需等价门的数目相加得到。对于全定制的实现, 设计大小可采用所有组件中晶体管的数目来近似, 或者采用组件布图和布局后边界框的面积来近似。

在本节中, 我们将讲述确定组件数量的技术, 这些组件是为实现功能描述所需。如 7.3.1 节所述的那样, 假定给定行为被实现为具有一个控制单元和一个数据通路的 FSM——对控制单元和数据通路的评估将分别讨论。作为一个例子, 我们还讲述了对于全定制实现的总布图面积的评估方法。同样的方法也可以应用到其他门阵列和 FPGA 技术中。

274

1. 数据通路

数据通路包含三种 RT 组件: 存储单元(如寄存器和锁存器)、功能单元(如 ALU 和比较

器)、互联单元(如多路选择器和总线)。数据通路综合的几种方法已经在[GDWL91]中讨论过了。在这一部分,我们将介绍用于评估这些组件的数量、大小和面积的技术。

存储单元用来保存行为中常量、变量和数组的数据值。最简单的评估策略是假设行为中的每个变量由独立的寄存器或存储器实现。但是,这样的实现可能会导致大量冗余的存储单元出现,因为在整个行为执行过程中好多变量的值是不需要的。通过将不同时使用的若干个变量映射到同一个寄存器或存储器,使存储单元数量达到最小化是可行的。为确定两个变量是否同时被使用,我们必须考虑它们的生存期。一个变量的生存期(lifetime)定义为从该变量的定义(写入该变量)到它的下一定义前的最后一次使用(读该变量的值)之间的时间间隔。一个设计所需存储单元的数量与交迭的变量生存期的最大数量相等。考察图 7-16a 例子中被调度的行为,包含从 v_1 到 v_{11} 这些变量。每个变量的生存期如图 7-16b 所示。变量 v_3 在状态 s_1 中计算为 v_1 与 v_2 之和,在状态 s_3 最后一次被加法操作使用以计算 v_8 。因此,变量 v_3 在状态 s_2 和 s_3 中是“活跃”的。

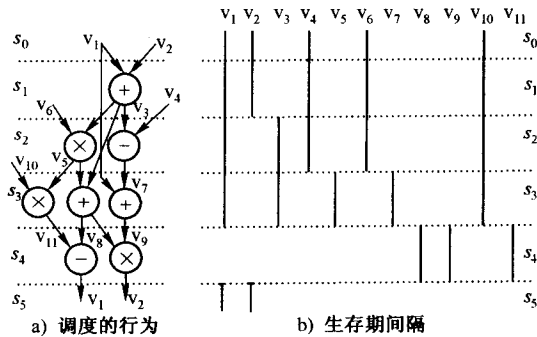


图 7-16 变量的生存期

一旦变量的生存期计算出来,就可采用若干种方法来确定实现这些变量所需的存储单元数。团划分(clique partitioning)方法可用于确定能映射到同一个存储单元的变量集合。图 $G = (V, E)$ 起源于给定的行为,在该行为中每个顶点 $v_i \in V$ 唯一表示一个变量 v_i , 而且仅仅当变量 v_i 和 v_j 可以存入相同存储单元,也就是说它们的生存期间隔互不交叉时,其对应的结点间也存在一条边 $e_{i,j} \in E$ 。 G 的一个完全子图或团(clique)表示该子图中所有变量的存储单元。为寻找最少数目的存储单元,我们必须将图 G 划分为最小数量的团,每个顶点仅属于一个团,该问题被称为团划分[CLR89]。

275

我们简要介绍解决团划分问题的一种启发式算法[CS86]。该方法包含若干迭代,每个迭代中具有最多公共相邻结点的一对结点被合并为一个结点。从其他结点到这两个结合的结点的边被到达新结点的边所代替。不断重复这样的合并结点过程,直至图中没有边,从而导致一组团的生成。每个团对应于一个存储单元,该单元将实现属于该团的变量。

我们下面针对一个行为来示范团划分方法,该行为中所有变量的生存期在图 7-16b 中示出。图 7-17a 示出该组变量的图模型,并适当地在顶点间添加了边。例如,由于 v_8 和 v_{10} 的生存期并不交叉,因此它们代表的顶点间存在一条边;而顶点 v_1 和 v_{10} 之间没有边是因为它们的生存期出现了交叉。因此,运用上述的团划分启发式方法,我们可以如图 7-17b 所示创建五个团并分别分配到寄存器 R1 到 R5。

276

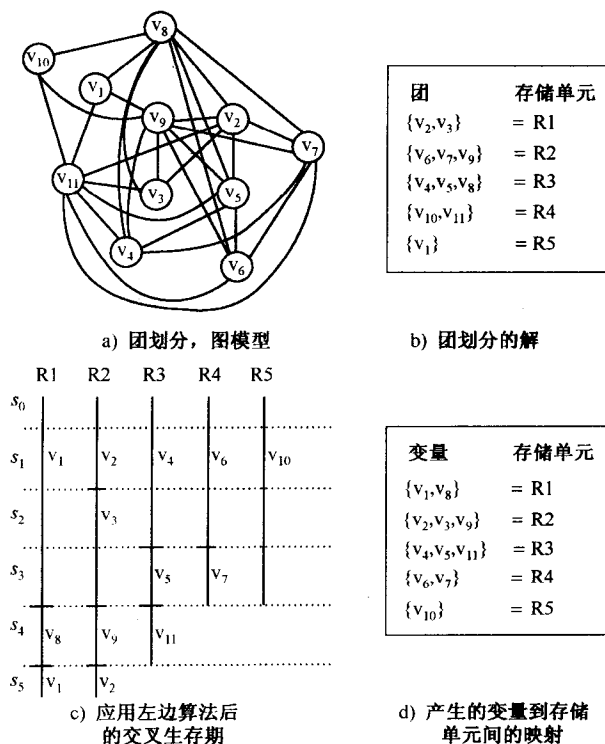


图 7-17 寄存器分配

另一个启发式团划分策略基于左边算法(left-edge algorithm)[HS71],目的是使 REAL 中寄存器的数量最小化[KP87]。行为中给定的一系列变量及其相应的生存期间隔,该算法首先按照这些变量生存期开始时刻的先后顺序将其按升序排列。算法对已排序的列表进行若干次扫描,并按它们在列表中的顺序来检查这些变量的生存期。在每次扫描中,该算法都分配一个新的寄存器并指派一新的变量到该寄存器,条件是该变量的生存期与已指派给该寄存器的所有变量的生存期不能出现交叉。这些被指派的变量随后就在变量有序列表中被删除。该过程重复执行直到所有变量全部指派到寄存器为止。

为了举例说明左边算法,我们将其应用于图 7-16 中被调度的行为。首先将这些变量按照生存期间隔来排序,产生序列 $L = (v_1, v_2, v_4, v_6, v_{10}, v_3, v_5, v_7, v_8, v_9, v_{11})$ 。在第一次扫描中,存储单元 R1 被分配并将变量 v_1 指派给它;扫描序列 L ,我们发现可以指派给存储单元 R1 的下一个变量是 v_8 ,因为它的生存期间隔与已指派给 R1 的变量 v_1 不交叉;而其他的变量的生存期都与变量 v_1 和 v_8 出现了交叉。因此,存储单元 R1 只包括变量 v_1 和 v_8 ,并将 L 中的 v_1 和 v_8 删除。在第二次扫描中,存储单元 R2 被分配并将 v_2 指派给它。扫描该序列 L ,可以发现变量 v_3 和 v_9 也可以指派给 R2。类似地重复执行,总共需要 5 次对 L 的扫描才能将所有变量指派给存储单元。图 7-17c 显示出左边算法的 5 次迭代过程中各个变量的生存期,而图 7-17d 显示了将变量指派给寄存器的最终结果。

比起假设每个变量配置一个存储单元的做法,上面的方法能对行为中所需存储单元数目做出更精确的评估。但是,以上两种方法的精确评估都是建立在大量额外计算的基础上的。

功能单元(functional unit)用来实现行为中的操作。对所需功能单元数量的评估可采用若干种方法。

278

首先,系统设计者需要明确说明功能单元的配置。

其次,如果行为已经按时间调度为几个控制步,那么可以采用团划分方法[CS86]来确定所需功能单元的数量。该方法包括图模型的构造,这与介绍过的确定存储单元数量的方法类似。图中每个结点代表行为中的一个操作。当且仅当相应操作被指派到不同的控制步中,而同时有某一功能单元能完成这两种操作时,两个结点间就会存在一条边。该图中的团表示可以被相同功能单元执行的操作集合。前一部分所述的启发性算法则可确定必要的功能单元的数量。

最后,如果为行为规定了性能约束(比如最大控制步数),实现行为所需的功能单元的最小数量可采用力指向算法(force-directed algorithm)[PK89b]来确定。该算法力图将相同类型的控制均匀分配至所有可得到的控制步中。这种均匀分配保证分配给一个控制步的功能单元在所有其他控制步中可以得到有效利用,并产生高的功能单元利用率。在每次迭代中,该算法仅分配一个未调度的操作至一个控制步中,以减少设计中预期的功能单元数量。当算法终止后,分配给任何控制步的特定类型操作的最大数量就是所需的该类型功能单元的需求数量。与团划分不同,力指向方法要求进行功能单元绑定(functional-unit binding),即将行为中的操作指派给特定的功能单元。这种操作到功能单元的映射具有本质意义,因为它能使我们确定存储单元与功能单元之间所需互联数量。

在行为中所有变量和操作都被映射到存储单元和功能单元之后,我们就可以对互联单元(interconnect unit)的数量进行评估了,如用于连接存储单元和功能单元的总线和多路选择器。用总线实现的优点是易布线,因为每条总线一般连接着一系列通信的单元。当一存储单元或功能单元使用总线读写数据时仅需要一很短的连接就可以接入总线。另一方面,用多路选择器实现可能导致堵塞,因为要求所有输入传送至一个多路选择器组件中去。

279

针对互联单元的评估可以分别从行为描述中和从变量和操作到寄存器和功能单元的映射中直接得到。

首先,确定设计中存储单元和功能单元之间的互联集合。例如,如果一行为具有语句 $A = B + C$,那么从变量 B 和变量 C 被分配的存储单元到执行加法操作的功能单元间就存在着连接,同样,功能单元到存储 A 的存储单元间亦有一条连接。

一旦这些连接已经确定,就需要将它们映射到一多路选择器或一总线中去。一个简单的策略是实现一组连接,该组连接具有共同的接收器,该接收器是指同一个多路选择器或一总线。图 7-18a 显示出了一组寄存器和一个功能单元的两个输入之间的连接。由于每个功能单元具有四个输入端,那么就需要两个 4×1 多路选择器 $M1$ 和 $M2$ 来实现该连接,如图 7-18b 所示。通过分解出至不同多路选择器的共同输入,并插入另外的多路选择器对这些共同输入选择其一,就可以达到降低多路选择器成本的目的。例如,图 7-18b 中的多路选择器 $M1$ 和 $M2$ 都具有来自于寄存器 $R2$ 、 $R3$ 和 $R4$ 的输入。从这些寄存器到功能单元的两个输入端之间的连接可以分解出来,由多路选择器 $M3$ 实现,如图 7-18c 所示。分解多路选择器输入的策略降低了设计中多路选择器输入的总数,也就降低了多路选择器的成本。例如,图 7-18b 中的两个多路选择器的输入总数为 8,而图 7-18c 中三个多路选择器的输入总数则仅为 7。

另一个评估连接单元数量的途径是采用团划分方法。这里要构造的图模型与前面介绍的

用以确定存储单元和功能单元数量的模型相似。图中每个顶点表示两个单元间的一个连接。两顶点间存在边当且仅当相应的连接并不在相同控制步中并发使用进行数据传输。该图中的每个团表示一个互联单元。一个团中所有顶点所代表的所有连接都将被分配至相同的多路选择器或总线上。

图 7-18d 显示出了在进行了团划分后,采用两条总线实现了图 7-18a 中的连接。当两条或更多总线需要被连接到相同的单元时,仍然需要多路选择器,从而导致两级互联结构。在图 7-18d 中,总线 B1 和 B2 都通过多路选择器 M1 连接到了功能单元的输入 I1。对于任何设计,存储单元与功能单元间的互联可能会采用多级互联单元实现。可采用总线或多路选择器来实现在任意一个级别上的互联。

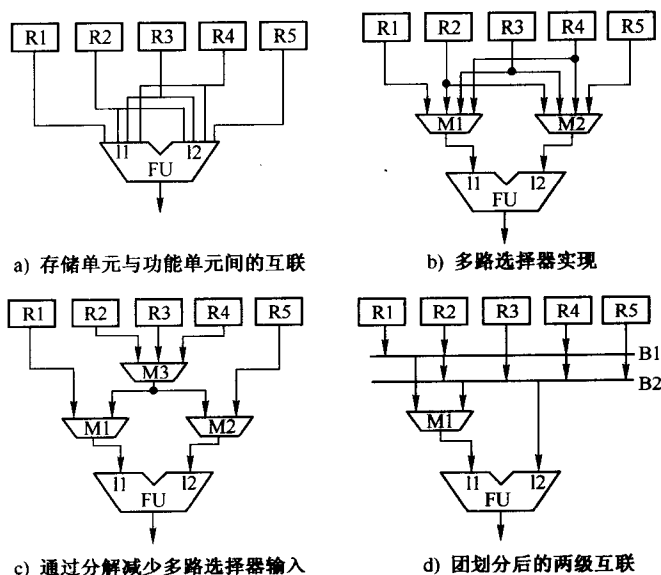


图 7-18 确定互联单元

到此已讲述了如何对数据通路中的存储、功能以及互联单元进行评估的方法,我们现在就来看如何对一个定制的 ASIC 实现来确定其数据通路的总面积。相同的公式经过简单地调整也可用于门阵列和 FPGA 的实现。

2. 数据通路面积的评估

数据通路的版图通常是一个位切片堆(bit-sliced stack)[WCG91],其中单独的数据通路组件放置成排,一个挨着一个,并使得每个单元的最低有效位(least-significant-bit, LSB)的位置对齐。该结构如图 7-19 所示。一个布线通道用于连接同一个位切片内的不同单元。控制线水平地布在第二金属层与电源层上,数据线垂直地布在多晶硅层或第一金属层上。

为确定数据通路的面积,我们需要确定一个位切片的总长度 L_{bit} 和高度 H_{bit} 。假设 L_{bit} 与每个位切片中晶体管数量成比例,我们首先来确定数据通路中每个位切片中的晶体管数量 $tr(DP)$ 。令 R 、 F 、 M 分别是为数据通路确定的寄存器、功能单元及多路选择器的数量。令 $tr(REG_i)$ 、 $tr(FU_j)$ 、 $tr(MUX_k)$ 分别表示位切片中第 i 个寄存器、第 j 个功能单元、第 k 个多路选择器中晶体管的数量,那么在每个位切片中晶体管的数量可以计算如下:

$$tr(DP) = \sum_{i=1}^R tr(REG_i) + \sum_{j=1}^F tr(FU_j) + \sum_{k=1}^M tr(MUX_k) \quad (7-20)$$

设 α 为以 μm /晶体管为单位表示的晶体管间距系数。对于一个给定的库, α 可通过求库中每个单元的宽度和晶体管数之比平均值来获得。如图 7-19 所示, 每个位切片的总长度可由下式求得:

$$L_{bit} = \alpha \times tr(DP) \quad (7-21) \quad [282]$$

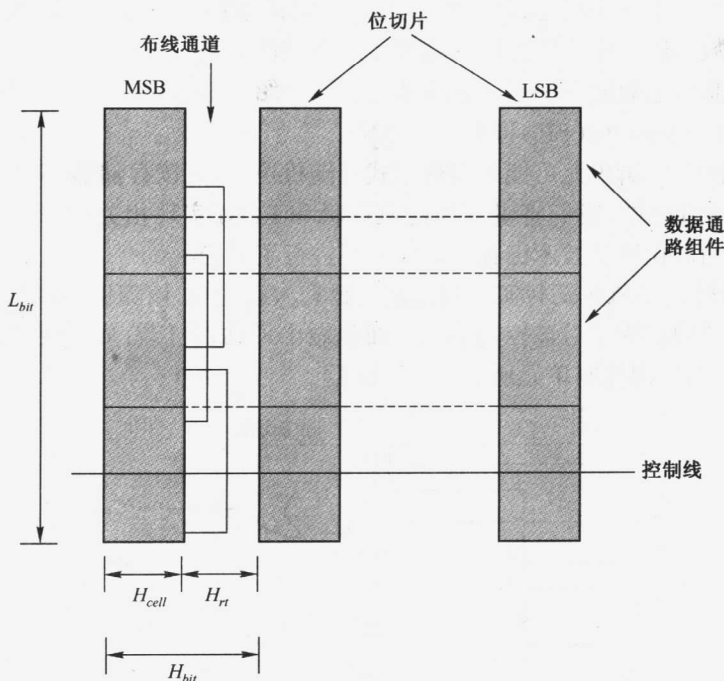


图 7-19 以位切片堆的形式排列的数据通路组件

在位切片中, 每个单元的高度 H_{cell} 是固定的且取决于库中标准单元的高度。与每个位切片相关的布线通道的高度 H_{rt} 则取决于为实现数据通路组件间的线网连接所需的布线轨道数。对每个位切片中所需轨道数的估计只能在该位切片中每个单元的位置确定之后才能获得。最小割算法(min-cut algorithm)[FM82]可用于该目的。所需的轨道数可估计为跨越垂直于通道的切割线的最大连接数。采用简单的布线算法可获得更好的评估, 如左边算法[HS71], 具有 $O(n \log n)$ 的复杂度, 其中 n 表示线网数。

一个较简单的方法, 需要知道同一轨道上能实现的平均线网数 $nets_per_track$, 该值可通过经验确定。设 β 为线间距, 即两条金属线之间的最小间隔。设 n 表示数据通路中组件之间的线网数。那么, 布线通道的高度 H_{rt} 可由下式计算:

$$H_{rt} = \frac{nets}{nets_per_track} \times \beta \quad (7-22)$$

这样, 每一个位切片的面积 $area(bit)$ 可以计算如下:

$$area(bit) = L_{bit} \times (H_{cell} + H_{rt}) \quad (7-23)$$

最后, 若 $bitwidth(DP)$ 表示数据通路组件的位宽, 那么数据通路总面积计算为:

$$area(DP) = bitwidth(DP) \times area(bit) \quad (7-24)$$

3. 控制单元面积评估

控制单元通过一系列的控制步或状态对设计确定时序,其中每一步表示一组能并发执行的数据通路操作。

控制单元包括状态寄存器、控制逻辑以及次态逻辑。状态寄存器用于存储经过编码后的状态。控制逻辑对数据通路组件定义了控制信号,其表现形式为当前状态的一个函数。次态逻辑定义了状态机的下一个状态,其表现形式为当前状态和数据通路状态线的一个函数。数据通路状态线将数据通路中比较操作的结果传送到控制逻辑。

评估行为所需状态数的方法在 7.3.3 节中有所介绍。如果一设计具有 N 个控制步,那么状态寄存器的位宽 $bitwidth(SR)$ 将会是 $\log_2 N$ 。

284

控制单元与次态逻辑单元可能的实现形式有随机逻辑、只读存储器(ROM)或可编程逻辑阵列(PLA)。在本部分中,我们将讲解与随机逻辑和 ROM 实现相关的评估技术。对用 PLA 实现的控制单元的面积评估技术已在[GDWL91]进行了讨论。

图 7-20 所示的是用随机逻辑实现的控制逻辑和次态逻辑,该随机逻辑是用与门和或门组成的二级网络实现的。为了对该控制单元的面积做出评估,我们首先需要确定与门和或门的数量和大小,以及为实现控制单元所需要的驱动器。

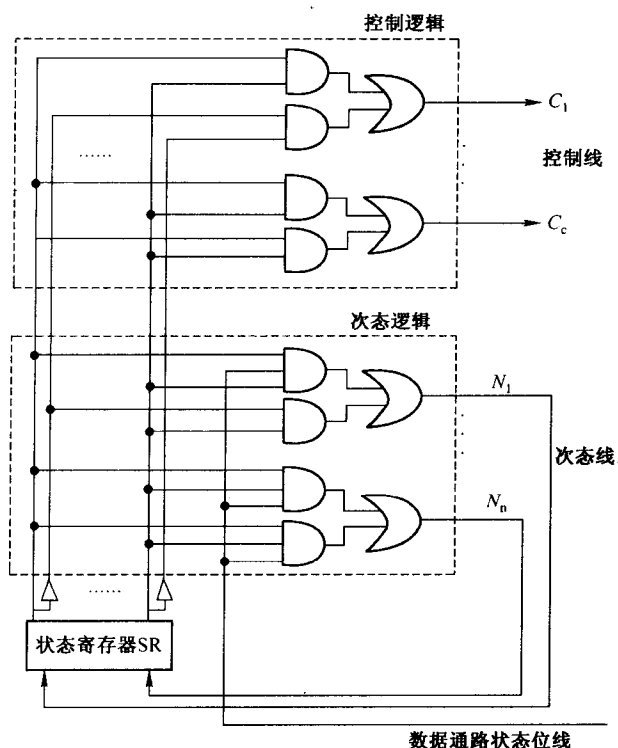


图 7-20 控制单元的两级与/或门实现

控制单元通常被描述成一个状态表,该表对数据通路组件中的每个状态定义了控制信号。为了确定控制单元所需的门的总数,可以为每个控制或次态列表产生布尔方程并对其进

行优化。但是,由于产生一个状态表、将布尔方程用公式表示、优化这些公式等过程都是非常耗时的,因此就需要另外的对控制单元面积进行评估的方法。我们现在就描述这样一个方法,该方法能对一个用随机逻辑实现的控制单元评估其所需要的门的数量和大小。

考虑图 7-20 所示的控制单元。由于每个控制线和次态线都需要一个或门,所以或门的总数应等于控制线与次态线的数量。对于一条控制线而言,一个或门的大小(输入端的个数)等于该控制线起作用的那些控制步的数目。例如,当某数据值置入寄存器时,用于驱动寄存器置入线的控制线 C_i 就起作用。 C_i 起作用的次数可以通过在系统描述中检查存储在该寄存器中的任何变量被赋予新值的次数得到。与此类似,只要一个功能单元要执行一个分配给它的操作,则使该功能单元动作的控制线就起作用。如果不止一个给定类型的功能单元被指派,那么某个控制线起作用的控制步数就可以估计为该种类型操作的数量对功能单元的数量求平均。考虑到选择线的二进制编码的情况,对于 $n \times 1$ 多路选择器的 $\log_2(n)$ 条选择线,以及该多路选择器为之选择数据的存储和功能单元,可以认为每个选择线在这些存储和功能单元的一半状态中是活跃的。为了为次态线确定或门的大小,我们假设每个次态线平均在设计中一半的控制步中进行了“翻转”。这样,驱动次态线的每个或门的大小就假定为控制步数的一半。

需要用与门对状态寄存器 SR 中的值所代表的当前状态进行解码。控制逻辑中与门的数量就是任何控制线或次态线起作用的控制步数的总和。与门总数的上界就是为该设计所确定的总控制步数。假设最多一个数据通路状态线影响一个次态线,则与门输入端数就近似为 $\text{bitwidth}(SR) + 1$,即多于状态寄存器的位宽。

此外,需要 $\text{bitwidth}(SR)$ 个单个位的驱动器,用来对状态寄存器的每个位的非反相线进行驱动,如图 7-20 所示。

在确定了或门、与门、驱动器及状态寄存器的大小和数量之后,我们就可以计算控制单元中晶体管的总数 $tr(CL)$ 。设 γ 表示晶体管的面积系数,单位是 $\mu\text{m}^2/\text{晶体管}$ 。系数 γ 可对于给定的标准单元库,并采用布局布线工具,通过实验的方法来确定。控制逻辑的总面积近似计算如下:

$$\text{area}(CL) = \gamma \times tr(CL) \quad (7-25)$$

以上方程并未考虑逻辑最小化和技术映射的影响。对于特定的逻辑库,对逻辑优化进行评估仍是一个尚未解决的问题。

控制单元的 ROM 实现如图 7-21 所示。对于一个 $W \times B$ 的 ROM, W 表示 ROM 中字的数量, B 表示每个字中的位数。

对于设计中的每个控制步,在 ROM 中都会存在一个字与之对应。这样,ROM 中字的总数 W 就等于为行为所确定的控制步数。对于每条控制线和次态线,ROM 中的每个字都包含一个位。因此,ROM 的位宽 B ,就是控制单元所产生的控制线和次态线的总数。

控制单元的面积可以计算为状态寄存器 SR 的面积与 $W \times B$ 的 ROM 的面积之和。

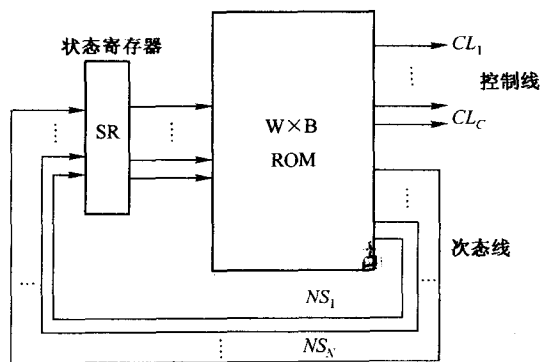


图 7-21 控制单元的 ROM 实现

7.3.7 引脚评估

一个给定的行为常常对该行为外的数据进行访问。这种通信将用一些连接该行为与被访问数据的连线来实现。如果每个行为都被综合到一个独立的芯片中,那么这些连线将成为芯片边界的引脚。

通信宽度(communication width)或在行为边界用于访问外部数据的连线数,取决于在功能描述中如何规定通信:

1) **端口声明**:端口的通信宽度要从行为中的端口声明来评估。例如,假定一个整数在设计中用 16 位的数来实现,那么以下的端口声明将导致在行为的边界,REQ 被评估为 1 位端口,ADDR 被评估为 16 位宽的端口,DATA 被评估为 22 位宽的端口。

```
port REQ : in bit;
port ADDR : out integer;
port DATA : in bit_vector(21 downto 0);
```

288

2) **通信通道**:通信通常抽象为一个进行数据传输的通道。通道可用硬件描述语言(HDL),如 HardwareC[KD88,DK88]、CSP(Communicating Sequential Processes)通信时序进程[Hoa78]以及 SpecCharts[NVG91a]来定义。通道将被综合到一个总线,该总线包含一组连线以及定义在这些连线上的数据传输操作协议。通信宽度是所有对总线有贡献的数据线和控制线的总和。例如,如果一个传输整数数据的通道用握手协议来实现,则需要 18 位来实现该通信(16 位用来实现数据传输,2 位控制信号用来实现握手协议)。

3) **全局数据**:通常系统描述中的行为需要访问变量,这些变量对其来说是可见的但却是外部的。例如,一个 VHDL 进程可能要通过访问全局信号来与另一个进程进行通信。对全局数据的访问在行为边界定义了隐含的端口。与明确声明端口的情况一样,通信宽度则取决于全局数据声明的类型。

4) **过程调用**:对行为要调用的过程的实现决定了过程与行为之间的通信宽度。如果在综合过程中过程是内嵌的,那么就不存在外部通信,也就没有必要在行为边界设置额外的引脚了。但是,如果该过程由一个独立的系统组件实现,那么所需的通信宽度就等于每个该过程参数(由参数声明类型来确定)所需通信宽度的总和。此外,还需要握手信号以允许主行为适时地开始或结束该过程。

对于给定的行为 B ,设 P 表示端口声明集合, C 表示通信通道集合, V 表示被访问的全局变量集合, S 表示被行为调用的过程集合。如果 $width(x)$ 表示访问目标 x 所需要的连线数量,那么行为边界处所需要的总互连线如下决定:

289

$$pins(B) = \sum_{p_i \in P} width(p_i) + \sum_{c_i \in C} width(c_i) + \sum_{v_i \in V} width(v_i) + \sum_{s_i \in S} width(s_i) \quad (7-26)$$

7.4 软件评估

在 7.2.2 节中介绍了软件质量度量。在介绍针对程序大小、数据存储大小和执行时间的评估方法之前,我们先来简要地讨论一下软件评估的基本模型。

7.4.1 软件评估模型

对于一个软件实现,行为描述必须编译成目标处理器的指令集。假设行为中的变量被映

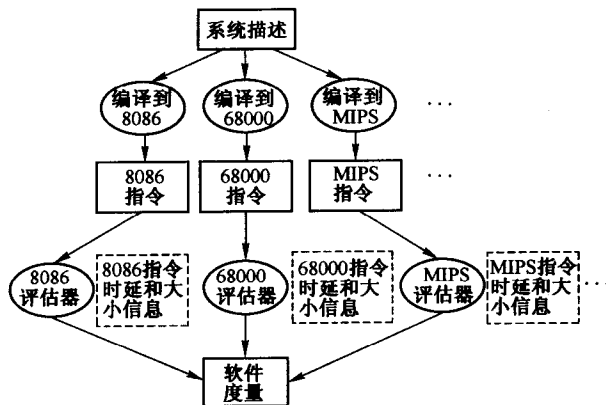
射到与处理器关联的存储器中。因此,所有对行为中变量的访问都假设对存储器的读/写操作实现。功能描述中的并发行为可在一个或多个处理器上实现。在用单个处理器实现的情况下,并发行为必须交错执行,以满足数据依赖或者时延限制。映射到同一个处理器上的两个或多个行为之间的通信是通过共享存储器地址来实现的。如果两个行为映射到两个不同的处理器上,它们既可通过共享存储器来通信,也可通过物理连接点直接通信。

软件评估可采用两种模型——专用处理器模型和通用模型。下面将分别介绍这两种模型。

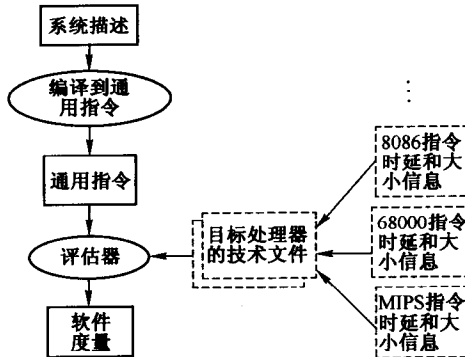
1. 专用处理器评估模型

如图 7-22a 所示,在专用处理器评估模型下,度量的准确值是通过采用处理器专用的编译器将每种行为编译到预期处理器的指令集中来计算的。根据与每个处理器关联的时延(timing)信息(例如执行每条指令所需要的时钟周期数)和大小(size)信息(如每条指令所需的字节数),我们可以确定被编译行为的大小和执行时间。例如,如果一个行为必须在 Intel 8086 微处理器上实现,首先要将它编译成 8086 指令集。获得已编译的行为后,就能用与 8086 指令集相关联的时延和大小信息来确定该软件执行的性能和大小。类似地,如果该行为要在 Motorola 68000 处理器上实现,就需要将其编译到 68000 指令集。基于 68000 指令的时延和大小信息,评估器就能获得该行为的软件度量。由于这种评估器是以某种专用处理器为目标,故称之为专用处理器模型。

290



a) 专用处理器评估模型



b) 通用评估模型

图 7-22 软件评估的方法

专用处理器模型的主要优点是获得的估计值很准确,因为行为是实际地编译到选定执行的处理器上的。但是,这种估量方法需要针对每种专用处理器的专用编译器。因此,已有的评估器很难适应一种新的处理器。此外,以评估为目的而编译一个行为需要很大的计算量。

2. 通用评估模型

专用处理器模型根据不同的目标处理器而使用不同的编译器和评估器,与此不同,在 [GGN94]中提出了一种通用评估模型。如图 7-22b 所示,在这种评估模型下,行为首先被编译成一组通用三地址指令。在这里可以使用专用处理器的技术文件,包含时钟周期数和每种通用指令所需字节数等信息。评估器根据通用指令和目标处理器的技术文件计算出软件度量。

291

通用指令集包含如下五类:

- 1) 算术/逻辑/关系指令: < *des* ← *src1* op *src2* >;
- 2) 移动/装入/存储指令: < *des* ← *src* >;
- 3) 条件转移指令: < if *cond* goto *label* >;
- 4) 非条件转移指令: < goto *label e* >;
- 5) 过程调用指令: < call *label* >;

在以上的指令中, *des* 表示一个寄存器或一个存储单元,而 *src* 和 *cond* 表示常量、寄存器或者存储器地址。 *label* 项表示过程名或者指令标号。除了以上这些指令,通用三地址指令集还包括 *return* 和 *null* 指令。

每个目标处理器的技术文件可以由处理器指令集的时延和大小信息中产生出来。图 7-23 显示出 < *dmem3* ← *dmem1* + *dmem2* > 型的通用指令所需时钟周期数的计算过程。这里的 *dmem* 指出是一种直接存储器寻址模式。通用指令首先映射成目标处理器的指令序列。该通用指令所需要的总时钟周期数是该指令序列中每条单独指令的时钟周期数之和。图 7-23 中的 *EA1* 和 *EA2* 表示在位移寻址模式下计算有效地址所需时间,在 8086 和 68020 处理器上,这两个数分别是 6 和 8 个时钟周期。这样,通用指令在 8086 和 68020 处理器上的运行实行时间分别是 35 和 22 个时钟周期。

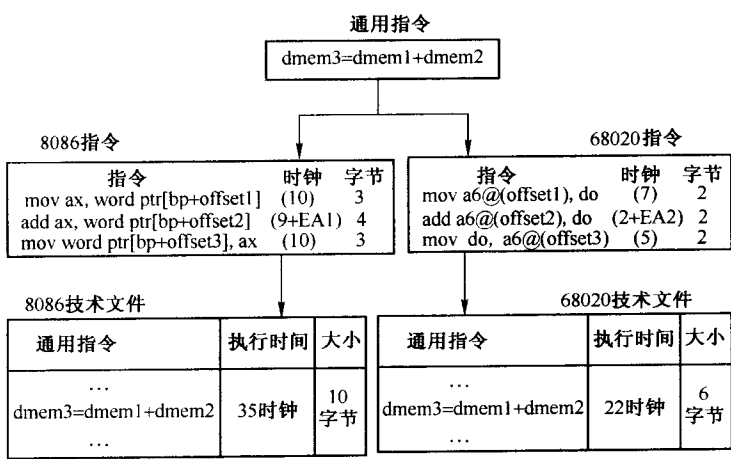


图 7-23 为通用指令获得技术文件

292

采用类似的方法,我们对于每种类型的通用指令,都可以得到其编译到 8086 或者 68020 处理器上时所需要的字节数。例如,图 7-23 中给出的通用指令可编译成三条 8086 指令组成的序列。三条指令的总大小为 10 字节,这个数可作为通用指令大小的参考值写入 8086 处理器的技术文件中。

与专用处理器模型相比,通用评估模型有若干优点。首先,通用模型不要求为不同目标处理器准备不同的编译器和评估器。软件评估要的只是一个编译器、一个评估器及一组技术文件。其次,通用模型使得将评估器重定向到新的处理器变得很容易。重定向的工作包括提供面向新的处理器的技术文件。在专用处理器模型中,则新的处理器除了需要处理器指令集的时延和大小信息之外,还需要一个编译器。最后,每种类型处理器的特点都会在该处理器的技术文件中体现出来。通用三地址指令摆脱了指令的特异性。因此,将行为编译成通用指令要比编译成作为处理器指令集一部分的复杂指令来得更简单更快速。通用模型的一个缺点是评估时准确率较低,这在很大程度上是因为通用指令集代表的仅仅是处理器集全部指令的一小部分。一组真正的处理器指令,比起与之等价的一组通用指令,应该能够更有效地实现规定的操作。

293

我们将采用通用评估模型来讨论软件实现中对程序存储器大小、数据存储器大小及软件性能的评估技术。同样的技术也可应用于专用处理器评估模型。

294

7.4.2 程序执行时间

一个行为的软件或程序的执行时间可采用动态模拟和静态评估两种方法之一来确定。给定一组输入数据,动态模拟执行程序并记录每次执行所需要的时钟周期数。给定不同组的输入数据,由于存在着与数据相关的条件分支和循环,动态模拟就会得到不同的时钟周期数。另一方面,静态评估对输入数据不敏感,而且,若循环迭代的次数已知并且条件分支的概率能准确预测,静态评估就能得出相当准确的结果。此外,静态评估比动态模拟更快,且需要的空间更小。在这个部分,我们将描述一种静态评估技术及其在软件性能评估方面的应用。

软件性能评估与 7.3.4 节中介绍的评估硬件执行时间类似。首先,将功能描述分成几个基本的块。在每个基本块中,语句集合被编译成通用三地址指令集合。每个基本块 b_i 的执行时间 $exectime(b_i)$ 可由在这个基本块内的每条通用指令的执行时间(目标处理器的技术文件中给出的该通用指令的时钟周期数)之和来决定。

接着,给定行为的基本块结构被映射到一个等价的控制流图 G 上。 G 上的每个结点有一个权值,该值与相应基本块的软件执行时间相等。通过应用基于概率的流分析技术,我们可以得到基本块在行为中的执行频率。最后,可采用方程 7-14 所示的方法来获得整个行为在平均情况下的软件执行时间。

一般情况下,编译器通过寄存器分配、循环优化和全局优化等技术来优化目标代码。设计者可以通过向编译器传送特殊标志位直接调用这些优化。本节中介绍的通用评估模型不使用任何优化启发式方法,因此软件执行时间的评估计算是针对非优化代码的。若要评估优化代码的性能,我们需要知道设计者用于产生机器指令的编译器的优化率。性能优化率(performance-optimization ratio) δ 定义为优化代码性能与非优化代码性能的比率。通过采用著名的 Livermore 循环内核进行的若干次实验,[GGN94]得出从 C 转成 8086、80286、68000 和 68020 指令集的优化率分别为 0.74、0.68、0.54 和 0.49。值得强调的是这些系数取决于所采用的数

295

据集合。因此,性能评价的精确度将取决于代码和数据集合与用于决定优化率的代码和数据集合的接近程度。

经过流分析之后,得到行为 B 的性能评价 $exectime(B)$,再乘上一个性能优化率 δ ,以考虑编译器优化的影响,也就是说,若设编译和优化过的行为的软件执行时间被评估为 $optimized_exectime(B)$,则有:

$$optimized_exectime(B) = \delta \times exectime(B)$$

(7-27)

对于简单的处理器,以上的评估方法能够很好地工作。但是,对带有指令和数据缓存的流水线 and 并行处理器的执行时间进行评估仍然是一个尚未解决的问题。

7.4.3 程序存储大小

程序大小评估的第一步也涉及将给定行为编译成通用指令。每种通用指令的大小在目标处理器的技术文件中做出了规定。根据每条通用指令的大小,一个行为的程序存储大小可以通过求该行为中的每条通用指令大小之和来计算。如果一个行为 B 被编译成一个通用指令的集合 G ,且 $instr_size(g)$ 表示通用指令 g 的大小,行为 B 所需要的程序大小可用下式计算:

$$progsiz(B) = \sum_{g \in G} instr_size(g)$$

(7-28)

如同 7.4.2 节中介绍的方法,考虑到编译优化的影响,还可以计算出平均优化率 θ 。从 7-28 式中得到的程序大小,可以乘 θ 从而得出对于已编译和优化行为的更准确评估。

7.4.4 数据存储大小

通过检查功能描述中的数据声明可以得到数据存储的大小。数据声明 d 的数据存储大小 $datasize(d)$ 由 d 的基类型大小和 d 中所包含的元素个数决定。任何数据声明的基类型都是在语言定义中不可再分的类型。例如,考虑以下这个声明:

```
variable X : bit;  
variable Y : array ( 9 downto 0, 15 downto 0 ) of integer;
```

变量 X 的基类型是位,且它的基类型元素的个数是 1;变量 Y 的基类型是整数,且它的基类型元素的个数是 160。

描述中提供了每个不同基类型的数据项在软件实现中所需的大小,这些都定义在一个查找表中。图 7-24 显示出一些 VHDL 语言中定义的基类型对应的数据存储大小。对于声明 d , $datasize(d)$ 可以通过将 d 的基类型大小乘以 d 中的基类型元素的数量来计算。例如,上面的声明 Y ,基类型是整型,从图 7-24 的查找表中可以查出,整型需要四个字节的存储空间。由于变量 Y 有 160 个元素,所以声明 Y 的总存储大小为 640 字节。

声明基类型	数据存储大小(字节)
位,布尔	1
位向量	$\lceil n/8 \rceil$ n 为位数
字符	1
整数,自然数,正数	4
实数	8

图 7-24 一些 VHDL 基类型的数据大小

在得到每个声明的数据存储大小之后,行为 B 的数据存储大小就可以通过将它包含的每个声明的数据存储大小相加来求得。设 D 表示行为 B 中声明的集合,则行为 B 的数据存储大小可用以下方法求得:

296

297

$$datasize(B) = \sum_{d \in D} datasize(d) \quad (7-29)$$

假定所有变量生存期都和行为的执行时间相等,也就是说两个变量不需要共享存储器地址。在这个前提之下,上面介绍的对数据存储大小的评估方法是准确的。如果一个行为含有大量短生存期的变量,则数据存储大小的评估就必须结合进生存期分析,以覆盖那些没有生存期重叠的变量。

7.5 系统级工具的评估技术

在这节中,我们将对在一些高层次综合和系统设计工具中使用的质量度量和评估方法进行综述。

7.5.1 BUD

BUD(自底向上设计, Bottom Up Design)[MK90]是作为高层次综合系统的一部分,进行全局分析和调度的系统。如 6.5.2 节所述, BUD 采用分级结群算法将单一行为的操作划分成群,并利用详细的物理信息来估算每个群的面积和性能,进而对整个设计进行估算。

BUD 的输入是行为的值追踪(Value trace, VT)[EST78]表示。此外,还提供行为的追踪文件,包含模拟过程中每个操作执行的次数。当确定平均循环时间时,我们可以从追踪文件中得到行为中操作执行的概率。BUD 还可以访问包含有详细信息的单元数据库,这些信息包括单元功能、时延、长度、宽度及功率耗散。BUD 评估器的输出是实现行为所需的面积和执行时间。

给定含有一组操作的群集合,首先评估每个群的面积。群中的每一类操作都分配到一个单独的功能单元。时钟周期的初值由系统设计者指定。各种操作通过列表调度器分配给各个控制步。采用生存期分析来确定每个群中寄存器的数量和大小。多路选择器是基于群中端口数的多少分配给群。群的宽度和长度是由单元的宽度和长度决定的,这些单元是选择用来实现寄存器、功能单元及多路选择器的。

一旦每个群的面积确定了, BUD 就采用 Zimmerman[Zim88]提出的一种布图规划技术来确定每个群的布局 and 方位。首先,行为被递归地划分,产生了对设计的一种树表示。树的叶结点是设计中单独的群。然后,从叶结点到根结点对树的表示进行遍历,产生群的所有可能的排列方案。形状函数用来表示任何群组合的高度和宽度的关系。根据形状函数可以算出在某个特定的高宽比条件下群的排列方案。该布图最终被调整以适应连接群间的布线区域。

数据通路确定下来以后,要重新计算时钟周期 clk ,方法是求在任何一个控制步内信号通过数据通路的最大时延。根据为 VT 提供的追踪文件,可以确定控制步 $i(1 \leq i \leq N)$ 执行的概率 p_i 。这样就能确定行为的平均执行时间如下:

$$exectime(B) = clk \times \left(\sum_{i=1}^N p_i \right) \quad (7-30)$$

BUD 中评估器的主要优势是在评估过程中结合进了详细的物理设计信息。例如,执行了布图规划,确定了布线区域,以及在计算时钟周期的过程中考虑到了布线时延等。因此,对时延和面积的评估可以认为是精确的。然而, BUD 中进行详细评估导致的结果是计算评估结果所需的时间可能很长。例如,就布图规划步骤本身而言, [MK90] 公布了一个秒数量级的执行时间。随着系统规模的增加,在 BUD 中进行设计空间探索将会耗费很大的计算量。另外,

BUD 不估算控制单元面积,而这些控制单元面积在某些设计中很可能是整个面积的重要组成部分。

7.5.2 Aparty

Aparty[LT91, Lag89, LT89]是一个结构划分器,它将一个系统的行为划分成多个部分,每个部分可能表示一个芯片或者芯片上的一个块。给定一个行为的 VT 表示,Aparty 利用一个多级结群技术来产生对行为的划分。

Aparty 中评估器的输入是一个结群的集合。每个群包含该行为的若干个操作。评估器还需要访问一个面积文件,该文件包含实现每个操作所需要的每个位的面积。

Aparty 通过计算每个群中所有功能单元和多路选择器的面积之和来得到该群的面积。对于每个群,群中每种类型的操作都会分配到一个功能单元。如果给定类型不只有一个操作映射到同一个群上,那么适当的多路选择器就会被分配给这些功能单元以实现这些操作。

Aparty 对每个群边界互联(连线的数量)的评估方法是计算该群与其他群之间连接的大小之和。

300

评估性能是通过采用 CSTEP 调度器[Nes87]对功能描述进行调度而产生的控制步数。

Aparty 的评估体现的是速度和准确度的平衡。通过采用一个只包含功能单元和多路选择器的简单模型,可获得对群面积的快速评估。因此,设计者就能探索更大的设计空间。但同时,评估还是相当粗糙的,因为寄存器、互联和控制单元面积都没有考虑进来。

7.5.3 Vulcan

Vulcan[GD90]划分工具将行为描述成一种图,这种图包含两种边,依赖边和排斥边。顶点表示该行为中的操作。依赖边表示操作间的依赖关系,而排斥边表示相应操作的排他性,这意味着在实现中这些操作可以共享相同的功能单元。每个顶点都带有相应操作所需的面积和表示传播时延的时延(用时钟周期数表示)。时钟周期被规定为 Vulcan 的一个输入。

图的面积估计可以简单地计算为该图中所有顶点面积代价之和。然而,由排斥边连接的顶点集合代表的是可以共享硬件的操作,所以在面积评估中只计入一个顶点。

图的性能可以通过计算图中从起点到终点的最长路径的长度来得到。也就是说,将最长路径上每个顶点的时延累加起来。如果一条边所连接的两个顶点被分配到不同的划分部分中,那么这条边上带有一个单位时延,以反映引入的通信代价。

Vulcan 中评估器的优点是它能够提供更快速的评估,因为面积和性能的评估都是通过对图中顶点所带的代价和时延简单求和得到的。面积评估的准确度可能比较低,因为布线、寄存器面积以及顶点对功能单元的共享等情况都被忽略了。

301

7.5.4 SpecSyn

SpecSyn[GVN94]是一个为设计者提供设计工具将系统描述划分成一组系统组件的系统设计框架。产生的系统组件表示芯片、内存、处理器上执行的软件或者总线。

SpecSyn 的输入是该系统可执行的系统描述。VHDL[IEE88]和 SpecCharts[NVG91a]语言可用于该目的。要评估诸如行为执行时间、时钟周期及面积等硬件参数,该工具提供了设计库(包含数量、类型、时延、面积及 RT 组件等用于实现设计的信息)。要评估诸如程序的执行时间和大小等软件度量,该工具提供了类似于 7.4.1 节所描述的处理器技术文件(包含指令大小和在目标处理器上的执行时间等信息)。

时钟周期评估器采用 7.3.2 节提出的松弛最小化方法选择时钟周期。SpecSyn 还允许设计者明确地指定时钟周期。

SpecSyn 中的面积评估允许设计者指派功能单元以实现某个行为。如果没有指派功能单元,就假设进行对每种功能单元的最少指派。控制步数的评估采用的是 7.3.3 节中描述的“操作符-使用”方法。寄存器的评估则采用生存期分析。由于评估器没有实际地将操作调度到控制步中,变量的生存期就以语句级的粒度来定义,即用语句数量来确定变量的定义和使用。这个方法不需要建立数据流图并且相对而言要快得多。接下来,要评估的是寄存器和功能单元所需多路选择器的数量和大小。鉴于对控制逻辑面积的评估采用类似于 7.3.6 节介绍的技术,故假设控制单元用一个状态寄存器和一个二级与或门网络来实现。

给定一个用 VHDL 串行语句描述的行为,SpecSyn 性能评估器首先确定每个基本块所需要的控制步数,可采用 7.3.3 节介绍的操作符-使用方法。使用基于概率的流分析方法将对每个基本块的评估综合起来从而获得行为的执行时间。

302

如果采用 SpecCharts 语言来描述一个层次化的行为,那么 SpecSyn 中的性能评估以一种至底向上的方式进行。每个叶行为的执行时间以与前面介绍的针对 VHDL 顺序语句相类似的方式来确定。对于任何层次级别上的任何行为,该行为执行时间的评估可通过将其子行为的评估结果结合起来的方法来确定。如果一个行为包含有并发的子行为,则该行为的执行时间是为它的那些子行为的执行时间所做评估的最大值。

SpecSyn 为设计者提供了评估器,用于评估引脚和行为对变量、过程、通信通道的访问次数。这些可由设计者用来决定将系统描述中的对象分配到不同的系统组件。

对于软件实现,SpecSyn 提供了对执行时间、程序大小和数据大小[GGN94]的评估器。平均和最坏情况下对软件执行时间报告的错误分别是 8% 和 19%。在程序存储大小评估方面,则平均和最坏情况下的错误分别是 5% 和 8%。

任何一个系统设计决策通常只对设计做一个或一些改变。一个这种决策的例子是在系统划分中将一个变量从一个系统组件重新放置到另一个系统组件中。如果在每个设计决策之后都要根据系统描述重新进行度量评估,将是很浪费时间的事情,尤其是考虑到对设计本身只做了增量式的改变。为了避免这种重复评估,SpecSyn 评估器考虑设计的任何变化,以允许对评估进行增量式修改的方式来保留设计信息。给定系统的可执行描述,评估器首先经历一个建立的阶段,在这个阶段中将产生包含相关设计信息的详细内部结构。对于任何后继的设计决策,通过更新内部结构反映设计决策并重新计算评估,就可以在一个恒定的时间内获得评估。SpecSyn 支持对于执行时间、硬件面积、引脚以及软件大小的增量式评估更新。这种方法的主要优点是

303

通过减少获得评估所需要的时间,使得那些可能要计算数千个划分的复杂划分算法得以运用。

SpecSyn 评估器的主要优点在向划分器提供评估的速度方面。SpecSyn 使用设计中各个方面的近似值,例如,在没有做任何布图规划的情况下将设计面积评估为晶体管数量的函数,或者在没有将操作绑定到特定的功能单元上的情况下对多路选择器进行评估。因此,就使得系统划分器能在一个相对较短的时间内能检查大量的选择。但是,SpecSyn 中的评估器是依赖于描述的。例如,一个单状态的功能描述可能包含条件分支。控制步评估器将为测试每一个条件都分配一个控制步,即使整个设计可以在一个控制步完成。评估器在评估控制逻辑面积时不做任何优化,并且忽略了连线的时延。

7.6 结论和发展方向

在系统级对巨大的设计空间进行探索,就需要对质量度量进行快速地评估。在本章,我们定义了在设计级与硬件和软件实现都有关系的一些质量度量:时钟周期、控制步、通信速率、软件和硬件执行时间、面积、引脚以及程序和数据存储的大小。我们描述了用于评估这些系统参数的技术。此外,我们还定义了用来评估任何一种评估技术的三项准则:准确性、速度和保真度。我们还讨论了一些在高层次综合和系统设计工具中采用的评估方法。

设计质量评估的未来工作需着重于若干方面,大体可分为三个广阔的领域:

(a) **优化**:评估技术需要进一步加强,把综合和编译工具产生的优化影响包括进来。例如,控制和数据通路组件上的逻辑最小化可以在相当大程度上减少设计面积和时延。类似地,编译器也进行许多优化,如死代码的消除、寄存器文件的优化及公共子表达式的消除。缺乏考虑这些优化影响的技术,将很可能导致对设计面积和软件执行时间等类似质量度量的过高估计。

(b) **新的度量**:随着系统设计变得越来越精确,需要定义新的质量度量以及相应的评估技术。例如,对于设计特性如功率耗散、可测试性、软硬件集成、可维护性和可制造性等都需要定义度量并且发展相应的评估技术。

(c) **新的结构特征**:评估中也要估算更加复杂的结构特征带来的影响。例如,软件评估技术就需要升级,以将 VLIW、流水线处理、指令预取、矢量处理及高速缓存等结构特征包含进来。硬件评估技术也需要增强以包括如复杂时钟方案(例如,多相时钟)等特征。

7.7 练习

- 如图 7-25 所示, $E(D)$ 和 $M(D)$ 分别表示四个设计实现的质量度量的估计值和实际测量值。计算所用评估方法的保真度。
- 为以下每个设计特性设计两个度量:
(a)功率耗散,(b)可测试性设计,(c)可制造性。
- 考虑图 7-7 所示的硬件设计模型。根据控制和状态位寄存器中存在的组件时延(不考虑任何线网时延),列出一个时钟周期的公式。
- 图 7-26 所示的是一个行为描述中各种操作出现的次数。图中还给出了实现这些操作所需功能单元的时延。使用最大操作数时延法和松弛最小化方法估算时钟周期,并计算这两种估算的时钟利用率。

设计点	$E(D)$	$M(D)$
W	112	109
X	128	137
Y	139	121
Z	205	132

图 7-25 某度量的估计值和测量值

操作	$occur(t_i)$	$delay(t_i)$
加	4	49
乘	9	163

图 7-26 操作出现次数和时延

- 如果一些操作用流水线功能单元实现,如何修改“操作符-使用”方法以确定控制步数?

6. 应用“操作符-使用”方法计算下面 VHDL 语句被调度所需要的控制步数。假设时钟周期是 25 ns,并采用一个乘法器(时延 100 ns)和两个加法器(50 ns)实现这个设计。

```
A := B + (C * 3);  
B := B + C + D;  
E := A * A;  
X := C + D + Y;  
D := A + E;
```

306

7. 设计两个优先级函数,在应用列表调度算法时将优先级列表中的操作进行排序。使用这两个新函数,调度图 7-12b 中的行为。假设可用的元件有:一个加法器、一个减法器 and 两个乘法器。
8. 如何修改 7.3.3 节中介绍的列表调度算法使之能考虑下面的每个因素?
- (a) 多周期操作;
 - (b) 流水线功能单元;
 - (c) 需要固定时延的访问内存。
9. 对于一个要在带有 cache(高速缓冲存储器)的处理器上实现的行为,设对 cache 每次访问的命中率(在 cache 中找到所需数据的概率)为 0.8。如果 cache 和主存的访问时间分别是 50ns 和 300ns,则该行为每次访问内存所需要的平均时间是多少?
- * 10. 假设对于一个行为中的每个操作,组件库中都有若干具有不同时延和成本的功能单元。在这些情况中,如何估计功能单元的数量?
- ** 11. 如何修改 7.4.2 节中介绍的估算软件执行时间的方法,使之能考虑 RISC 处理器中指令的流水线操作。
- ** 12. 给定一个行为描述,该行为将在带有 cache 的处理器上以软件方式执行,设计一种策略估算 cache 的访问命中率(在 cache 中找到所需数据的概率)。

307

第 8 章 设计描述细化

在第 2、3、4、5 章,我们介绍了一些用于获得系统功能描述的方法。在第 6 章和第 7 章,我们介绍了若干将这些功能划分到不同的系统组件上的技术,以及对这些划分结果的设计质量进行评估的技术。本章,我们将把注意力放在如何为我们选取的划分结果和所选组件创建一个细化的描述上。至此,我们将向着系统的最终实现又迈进一步。

8.1 引言

一个系统描述包括很多功能对象(functional object),如:行为、变量以及通信通道。在系统设计阶段,我们将这些功能对象结群起来,成为一个群组,并将它们描述成为一组系统组件(system component),如:处理器(processor)、专用集成电路(ASIC)、内存(memory)和总线(bus)。虽然功能对象是没有任何结构的,但是系统组件拥有很好的结构信息,如,一个芯片的引脚数、内存的字(word)数和大小、总线中的线(wire)数等。将系统描述中的功能对象映射成为系统组件的过程称为描述细化(specification refinement)。

描述细化是系统设计中一个重要的部分,这是因为:首先,描述细化通过更新系统描述来反映系统划分时所决定的结群策略,从而使系统描述在各个方面保持一致。例如:当几个变量被结群在一起,并且被分配到同一个存储器上,那么,对这些变量的声明和引用都必须被更新。描述细化工作将会把变量声明替换成为存储器中数组的声明,还会把所有对原有变量的引用更新为对数组单元的访问。其次,细化使得系统描述可模拟,使得设计者可以在系统设计步骤完成后对系统功能的正确性进行验证。最后,细化后的描述可以在系统设计之后作为验证、综合和编译工具的输入。

309

本章中,我们将介绍一系列描述细化的工作。首先,我们将介绍与变量和通道组的具体实现相关的细化工作。其次,我们介绍一些机制,用于解决由多个行为并发访问共享资源所导致的冲突,如多个变量被结群到同一个存储器或者多个通道被结群到同一个总线。再次,我们讨论将行为分配到有固定接口(引脚结构和通信协议)的标准组件时可能产生的影响,并介绍将两个接口不兼容的标准组件连接在一起的方法。最后,我们讨论已经分配到硬件和软件组件上的行为之间的通信实现的相关内容。

8.2 细化变量群组

如果一组变量被映射到一块存储器,这块存储器在细化描述中的模型就将是一个有相应大小和位宽的数组。变量群组(variable grouping)描述的细化工作包括两部分:变量折叠和存储器地址转换。

8.2.1 变量折叠

系统划分把一组变量映射到一块分配好的、有固定的字(word)数和位宽的存储区域。不同的变量可以有不同的大小(变量的位数)。变量折叠的任务就是将变量中的位对应到存储器中每个字(word)的位(bit)上。

310

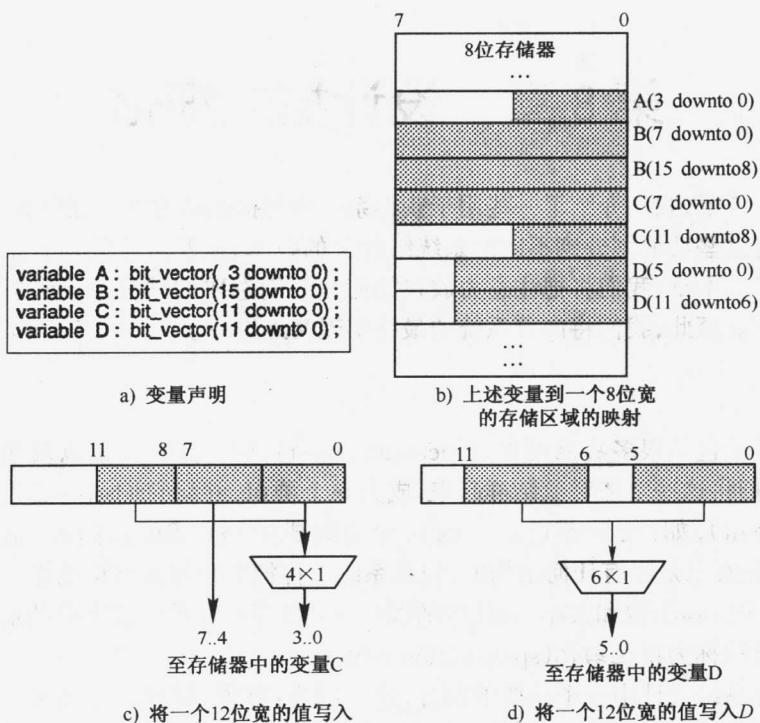


图 8-1 存储位宽映射

如果变量的位宽比存储器的位宽还小,或者正好是存储器位宽的整数倍,那么变量折叠工作就非常简单。如:图 8-1a 中的变量声明必须被映射到一个 8 位宽的存储器上,如图 8-1b 所示,变量 A 和 B 被分别分配了 1 个和 2 个存储器字。

如果变量的位宽不是存储器位宽的整数倍,则有两种映射方式。对于 12 位宽的变量 C,低 8 位被分配了一个存储器地址,而高 4 位则被映射到连续的下一个地址,如图 8-1b 所示。对于另一个 12 位宽的变量 D,低 6 位被映射到一个地址,而高 6 位被映射到连续的下一个地址。第一种方法在将变量 C 写入存储器的时候需要的多路选择器比较小,但是需要有 8 个总线驱动器连接在存储器数据总线上,如图 8-1c 所示。后一种方法需要一个较大的多路选择器,但是只需要 6 个总线驱动器连接在存储器数据总线上,如图 8-1d 所示。

8.2.2 存储地址转换

为已经映射到存储器的变量分配地址和修改所有的对这些变量的引用的工作称为存储地址转换。

存储器地址的分配和变量的处理顺序无关。每个变量需要的存储字数可采用变量折叠方法确定。一个数组变量中的元素必须分配在一段连续的存储区域中。

对于标量变量,存储地址的变换相对直接。假设一个标量 V,它属于一组变量,这组变量表示为数组 MEM,而且 V 被分配在 MEM 中地址 30 的位置。系统描述中所有对变量 V 的引用都被简单地替换成为 MEM(30)。

对于数组变量,如果它和别的变量被组合在一起,那么这个变量的索引地址必须在细化这个变量的引用的时候进行更新。例如,如果对于一个数组变量 V(63 downto 0),它被分配到 MEM 的 100 到 163 地址,我们在更新对 V 的引用的时候必须考虑如下情况:

首先,如果数组变量 V 是完全通过立即数索引的,那么对它的引用可以简单地替换成为对 MEM 的引用。这样, $V(0)$ 就替换为 $MEM(100)$,而 $V(36)$ 被替换为 $MEM(136)$ 。

其次,如果该数组是通过包含变量的表达式索引的,那么,相对于数组变量第一个元素的初始存储器地址的一个偏移量必须被加在表达式中。例如,一个数组变量 V ,用 K 和 J 作为索引,如图 8-2a 中代码所示。由于 V 被分配到 MEM 中 100 开始的地址,那么 $V(J)$ 被替换为 $MEM(J + 100)$,而 $V(K)$ 被替换为 $MEM(K + 100)$,如图 8-2b 所示。

<pre> variable J, K : integer := 0; variable V : IntArray (63 downto 0); ... V(K) := 3; X := V(36); V(J) := X; ... for J in 0 to 63 loop SUM := SUM + V(J); end loop; ... </pre>	<pre> variable J, K : integer := 0; variable MEM : IntArray (255 downto 0); ... MEM(K + 100) := 3; X := MEM(136); MEM(J + 100) := X; ... for J in 0 to 63 loop SUM := SUM + MEM(J + 100); end loop; ... </pre>	<pre> variable J : integer := 100; variable K : integer := 0; variable MEM : IntArray (255 downto 0); ... MEM(K + 100) := 3; X := MEM(136); MEM(J) := X; ... for J in 100 to 163 loop SUM := SUM + MEM(J); end loop; ... </pre>
--	--	---

a) 原始描述

b) V 的所有索引表达式加入
偏移量后的描述

c) 更新索引变量/后的描述

图 8-2 存储器地址转换

再次,如果一个变量只用来索引一个数组,那么,可以通过对地址变量进行适当的初始化来避免添加偏移量的工作。例如,在图 8-2a 中,变量 J 只用来索引数组变量 V ,图 8-2c 说明如何通过初始化变量 J (包括在 for-loop 循环中)来消除对添加偏移量的需要。然而,如果变量 J 用来索引多个数组,本方法则可能不再适用。在这种情况下,我们可以把一个数组分配在 0 地址开始的存储区域中,从而至少减少一次存储偏移量添加工作。

312

8.3 通道细化

在一个系统描述中,并发行为通过在抽象的通信通道上发送信息彼此进行通信。为了减小互联代价,系统中的通道会被结群,同一个群中的通道用同一个物理媒介实现,称为总线(bus)。一条总线包含一组连线,实际的数据传输就是在这些连线上按照一个总线协议实现的。为每组通道生成总线以及传输协议的工作就称为接口细化。在本节我们就讨论接口细化的方法。本节中的“行为”是指一组并发行为中的一个,通常称为一个进程。

313

8.3.1 通道和总线的表征

对于任何通道,都只有一个主(master)行为发起和控制数据传输,同时有多个从(slave)行为对主行为发起的通信做出响应。如果主行为通过通道发送(或接收)数据,则通道的相应方向为写(或读)。通道通常是单向的,这意味着如果一个行为要对另一个行为中的某个变量既要读又要写,就需要为每个方向的数据传输建立单独的通道。

通道由 4 个参数来表征。评估这些参数值的方法见第 7 章。通道数据大小(channel data size) $bits(C)$ 表示通过通道 C 传输的单个消息的位数。数据大小包括为了通过通道访问数组变量所需要的任何地址位。访问数(number of accesses) $access(P, C)$ 表示行为 P 在其生命期内通过通道 C 传输数据的次数。通道平均速率(channel average rate) $avgrate(C)$ 是通过通道 C 进行通信的行为在其生命期内进行数据传送的平均速率。通道峰值速率(channel peak rate) $peakrate(C)$ 是单个消息在通道 C 上传输的速率。

用于实现任何通道或者通道群的总线都可用4个参数来表征。总线宽($buswidth$) $buswidth(B)$ 是总线 B 中的数据线的数目,行为间的消息通过这些数据线进行传输。每条总线都有一个相应的协议,该协议定义了一系列操作以实现数据在这些数据线上的传输。协议时延(protocol delay) $protdelay(B)$ 是在总线上进行一次数据传输时协议所占的总时延。平均总线速率(average bus rate) $avgrate(B)$ 在系统的总生命周期内,数据在总线上的平均传输速率。总线峰值速率(peak bus rate) $peakrate(B)$ 是数据通过总线的最大传输速率。总线峰值速率和总线宽之间有如下关系:

$$peakrate(B) = \frac{buswidth(B)}{protdelay(B)} \quad (8-1)$$

314

8.3.2 问题的定义

给定一组抽象通信通道,接口细化确定实现这些通道的总线宽和协议。接口细化通常由两个相互矛盾的目标来驱动。首先,它试图通过减小总线宽 $buswidth(B)$ 来最小化系统组件之间通过总线的互联代价。其次,它寻求通过增加总线的峰值速率 $peakrate(B)$ 来最大化总线的通信性能,从而增加总线的宽度 $buswidth(B)$ 。

接口细化包含两个任务:总线生成和协议生成。给定一组约束,总线生成确定总线的宽度,该总线用于实现通道组。当总线宽度确定后,协议生成选取并且生成通信协议,该协议最终实现数据在总线上的传输。我们将分别讨论总线生成和协议生成。

8.3.3 总线生成

本节中,我们将介绍在实现一组通道的过程中确定总线线宽的方法。

在[FKCD93]中介绍了一种简单的总线宽生成方法,在这种方法中,所有的被结群在一起的通道都有相同的消息大小。在这种情况下,通道是以这样的方式进行合并的,即在同样两个行为之间通信的时候,任何通道组中的所有通道都应该在时间上是独占使用的。因此,每个通道组就采用与组中任何通道大小相同的总线线宽来实现。

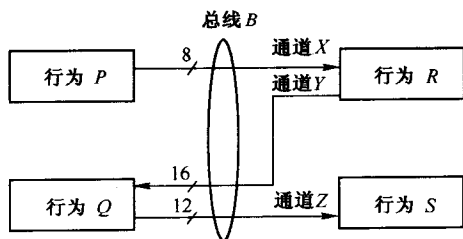


图 8-3 典型的总线, 通过将若干行为间传输不同大小数据的通道合并而成

在更一般的情况下,行为通过结群在一起的通道进行通信的时候,希望能够通过共享的物理介质进行同时通信。另外,不同的通道也可能在行为间传递不同大小的消息。这种情况在图 8-3 中进行了展示。行为 P 、 Q 、 R 、 S 通过通道 X 、 Y 、 Z 进行通信。通道 X 、 Y 、 Z 在系统划分的时候已经被划分在一起,并且用一个单独的总线 B 实现。三个通道通过总线分别传递三种不同大小的数据,即分别为 8 位、16 位和 12 位。此外,行为 P 向行为 R 传递数据的时候,行为 Q 也需要向行为 S 传递信息。这种针对一般情况的总线生成方法在[NG94]中有所介绍。我们下面详细讨论这种方法。

315

1. 确定总线速率

让我们来考虑两个通道, X 和 Y , 分别传递 8 位和 16 位消息, 如图 8-4 所示。每个消息传

递的位数标示于消息上方。为简单起见,我们假设图中所示的4秒时间间隔代表了行为在整个生命周期中通过通道X和Y进行通信的信息传递状况。可以看到,通道X和Y的平均速率分别是4位/秒和12位/秒。如果X和Y被合并到一条总线B中,那么总线的速率需要至少达到16位/秒,这样才能满足这两个原始通道的需求。通过通道传递的每个信息都已经在图上进行了标示,以易于与共享总线的传输数据相关联。考虑 $t=1$ 秒时刻在原始的Y通道上传递的消息Y2,现在要在总线B上在 $t=1.5$ 秒时刻进行传递。虽然单个消息的传递可能因为总线访问冲突而时延,但在通道合并之前通过单个通道传递的总位数仍在同样的时间内通过共享总线进行传递。

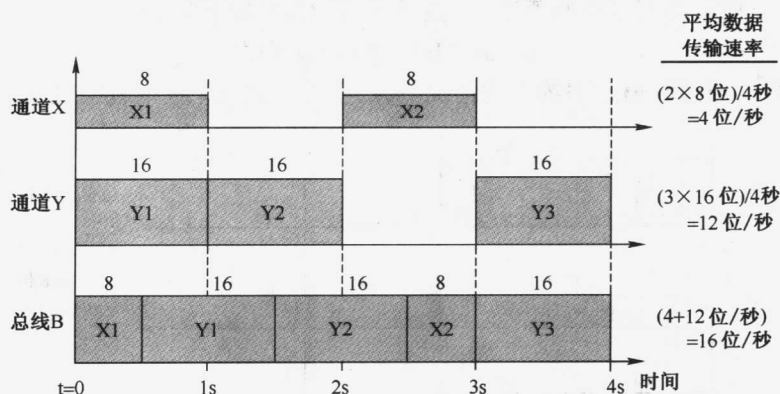


图 8-4 将通道 X 和 Y 合并到总线 B

在对如图 8-4 所示的总线 B 进行综合的时候,我们应该充分利用这样一个事实:一条通道不会总是在传输数据。如果我们将总线综合成为固定传输速率的,那么一个通道的空闲时间就可被别的通道利用来进行数据传递。

在将通道合并到一条总线上之前,如果通道在某个平均速率下进行数据传递,那么应该在总线上以同样的平均速率进行数据传递。这个目标能达到的条件是:总线 B 的平均速率 $avgrate(B)$ 必须大于单个通道的平均速率之和。这样:

$$avgrate(B) \geq \sum_{C \in B} avgrate(C) \quad (8-2)$$

总线生成的目标应该是:在以最少数量的连线和以公式(8-2)确定的平均速率条件下,进行总线综合。最有效的总线实现是:总线上永远没有空闲时间,而且总线总是以一个固定的速率传输数据。在这种情况下,总线的峰值速率和平均速率是一样的:

$$peakrate(B) = avgrate(B) \quad (8-3)$$

2. 总线生成的约束

对于一组被结群在一起,即将被实现为一条独立总线的通道来讲,可针对某些总线和通道的参数来规定约束和相对的权重。

最小/最大总线宽约束可能来自于为模块或芯片所规定的总引脚约束,通过总线进行通信的行为映射到这些模块或芯片上。

通道平均速率的约束可规定为确保行为不会因为在总线上的通信时延所减慢。给定一个通过若干通道进行通信的行为的执行时间约束,设计者可为在各种通道上进行通信分配或预留时间,通道的最小平均速率约束可由此得到。当某个通过通道进行通信的行为代表一个慢

速器件,即不能以快于某个速率来发送和接收数据时,就可由此规定最大通道平均速率。

在某些情况下,可规定最小通道峰值速率,以确保通道上某个消息的传递不占用过多时间。例如,考虑图 8-5a 中的通道 X,在前两个时间片($t=0$ 和 $t=1$)的每一个中都传递 16 位消息,剩下的两个时间片由通过通道 X 通信的行为用来进行内部计算。如果我们想用一条总线实现通道,根据公式 8-2,则总线平均速率则为 8 位/秒,产生图 8-5b 的运行状况。现在行为就需要 4 个时间片来进行总线通信。这显然是不能接受的,因为还需要额外的 2 个时间片来执行行为的内部计算(在图 8-5a 中,原来在时间片 $t=2$ 和 $t=3$ 中执行)。如果通道 X 的最小峰值速率被规定为 16 位/秒,我们就会得到如图 8-5c 所示的总线实现,而不需要附加任何时间片。因此,对于所有具有最小峰值速率约束的通道 C,有如下公式:

$$\text{peakrate}(B) > \text{peakrate}(C) \quad (8-4)$$

在规定了最小通道峰值速率约束的情况下,产生的总线在某些时候会是空闲的。

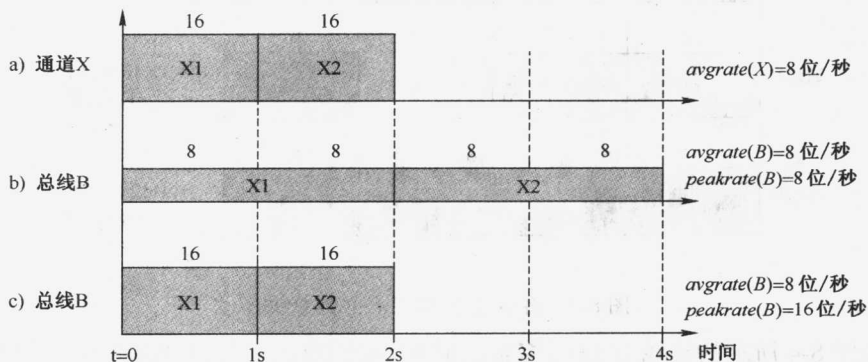


图 8-5 通道峰值速率约束: a) 通道 X 的执行跟踪, b) 无通道峰值速率约束下对总线 B 的综合, c) 在 16 位/秒的通道峰值速率下对总线 B 的综合

3. 确定总线宽的算法

前面,我们已经介绍了总线生成过程中可能规定的约束,现在我们就提出用于确定总线宽的算法[NG94]。算法假定数据线和控制线是分离的。在任何给定时刻,只有一个通道能通过总线传输数据。如果总线的线宽大于地址和数据的位置,那么地址和数据就会同时通过总线进行传递,否则,它们就要分两次分别传递。在后一种情况下,地址位必须被接收行为锁存。如果总线宽小于要传递消息的大小,那么消息就会分若干次进行传递。

所有通过总线中的一个通道进行通信的行为都假定是同步实现的。因此,通过总线传递一个消息需要整数个时钟周期。通过总线访问的变量被建模成为一个独立的行为,该行为响应其他行为的请求,通过总线发送和接收数据。这样,当计算一个通过总线访问变量的主行为的执行时间时,就假定具有这个变量的从行为永远都是准备好进行数据传输的,也就是说,通过总线访问变量没有同步时延。

总线宽生成算法的输入包含一组要被实现在一条总线上的通道,以及通道传输速率约束和总线宽约束。算法的输出是实现该通道组的总线宽度。

总线宽生成算法检查一系列可能的总线宽。对每个总线宽,计算总线峰值速率和每个通道的平均速率。在综合一条持续不断地传输数据的总线时,公式 8-2 和公式 8-3 要求总线的峰值速率要大于通道的平均速率之和。每个满足以上条件的总线宽都是一个可行的总线实

现。对这组可行的总线实现,每个都对应一个不同的总线宽,我们要从中选择一个代价最小的。如果没有约束,就选择单位总线宽,这对应于顺序的数据传输。

总线宽生成算法如算法 8.3.1 所示。首先,确定算法要进行检验的总线宽范围。最大总线宽 $maxwidth$ 就是通道上传递的最大消息的大小。最小总线宽 $minwidth$ 就是 1。

算法 8.3.1: 总线宽生成

```

if no constraints specified then
    return(1)
end if
/* 计算总线宽的范围 */
minwidth = 1
maxwidth = Max(bits(C))

mincost = ∞
mincostwidth = ∞
for currwidth in minwidth to maxwidth loop

    /* 计算总线峰值速率 */
    peakrate(B) = currwidth ÷ protdelay(B)

    /* 对 currwidth 计算通道平均速率之和 */
    avgratesum = 0;
    for all channels  $C \in B$  loop
        
$$avgrate(C) = \frac{access(P, C) \times bits(C)}{comptime(P) + commtime(P)}$$

        avgratesum = avgratesum + avgrate(C);
    end loop

    if (peakrate(B) > avgratesum) then
        /* 可行解, 确定最小代价 */
        currcost = ComputeCost(currwidth)
        if (currcost < mincost) then
            mincost = currcost
            mincostwidth = currwidth
        end if
    end if
end loop

if (mincost = ∞)
    then return(failure)
    else return(mincostwidth)
end if

```

变量 $currwidth$ 表示当前正在被算法估计的总线宽。对于每个在 ($minwidth$,

$maxwidth$)范围内的 $currwidth$ 值,总线峰值速率用公式 8-1 计算。

估算通道平均速率的方法已经在第 7 章进行了介绍。为简单起见,假设行为 P 只有一个通道 C 用来传递消息。当前总线宽小于消息位数的时候,需要若干次($\lceil \frac{bits(C)}{currwidth} \rceil$)传递才能完成一个消息的传递。行为 P 的通信时间可以通过公式计算:

$$commtime(P) = access(P, C) \times (\lceil \frac{bits(C)}{currwidth} \rceil \times protdelay(B)) \quad (8-5)$$

使用上面计算的 $commtime(P)$ 值,就可以用公式 7-18 来估算每个映射到总线的通道的平均速率。对于 $currwidth$ 的每个特定值,通道平均速率的总和保存在 $avgratesum$ 中。

如果总线的峰值速率,用 $peakrate(B)$ 表示,比所有通道平均速率的总和还小,那么 $currwidth$ 就是一个不可行的总线实现。于是在 $(minwidth, maxwidth)$ 中选取另一个较大的总线宽,再计算总线峰值速率,以及通道速率的总和。

如果总线峰值速率比各个通道速率之和,那么 $currwidth$ 表示一个可行的总线实现。对于任何为总线规定的总线宽、通道平均和峰值速率的约束, $ComputeCost$ 过程计算可行的总线实现的代价:每个约束的违反值的平方再乘上一个对该约束规定的相对权重,然后将所有这些加权的平方值求和。例如:假设只规定一个约束,就是限定总线宽的最大值为 $maxwires$ 。假设 k 表示这个约束的相关权重。对任何总线宽的值 $currwidth$,总线的代价函数定义为:

$$cost = \begin{cases} (k \times (currwidth - maxwires))^2 & currwidth > maxwires \\ 0 & \text{其他} \end{cases} \quad (8-6)$$

还可定义其他的代价函数用来估算特定总线宽的代价。例如,如果我们希望总线用最少数量的连线实现,那么代价就是 $currwidth$ 。

如果对通道组有多个可行解,就选择代价最低的总线宽来实现总线。算法 8.3.1 中的变量 $mincost$ 表示所有可行实现的最小代价,而变量 $mincostwidth$ 则表示对应于最小代价的总线宽。

如果在检查了所有的总线宽后都没有找到可行解,那么实现这组通道是不可行的。这组通道的任何一种实现都会逐渐造成通过总线通信的行为的时延。这种情况产生于当我们把若干平均速率很高的通道结群在一起,用一条总线来实现的时候。一种解决方法是将这组通道拆分开来,用多条总线实现。另外一种方法是从不可行解中选一个代价最小的。

通道 C	行为 P	访问变量	$bits(C)$	$access(P, C)$	$comptime(P)$ (clocks)
ch1	P1	V1	16 data + 7addr = 23 位	128	515
ch2	P2	V2	16 data + 7 addr = 23 位	128	129

图 8-6 温度控制器两个通道的消息大小、通道访问次数及计算时间

4. 总线宽度生成的一个例子

我们以一个温度控制器为例来说明总线宽度生成算法。控制器包括两个输入,用于感知房间内的温度和湿度,并检测四组规则来控制空调的操作。系统划分将行为和数组变量映射到两个不同的系统组件中,这样就在两个组件之间产生了几条通道。温度控制器的两个行为

$P1$ 和 $P2$ 分别通过通信通道 $ch1$ 和 $ch2$ 访问两个数组变量 $V1$ 和 $V2$, 这两个通道被结群在一起用一条总线 B 实现。图 8-6 列出温度控制器这两个通道的通道位数、计算时间以及通道访问次数。为简单起见, 所有的执行时间和数据传输率都用控制步表示。现在我们可以用总线宽度生成算法来确定由合并两个通道所形成总线的宽度了。假设总线实现的唯一约束是通道 $ch2$ 的最小峰值速率即 10 bits/clock(位/周期), 其相对权重是 10。

首先, 我们确定要检验的总线宽度的范围。 $ch1$ 和 $ch2$ 都要访问 128 个字(7 个地址位)的数组变量, 每个字的长度是 16 位。这样, 任意通道上的任何一次传输所需的最大位数就是 23。总线宽度的检验范围就是:

$$minwidth = 1 \text{ 和 } maxwidth = 23$$

我们从 $currwidth = 18$ 计算总线和通道的速率。假设采用完全握手协议来实现总线, 而且相关的协议时延是 2 个时钟周期, 即 $protldelay(B) = 2 \text{ clocks}$ 。根据式 8-1, 有:

$$peakrate(B) = 18 \div (2) = 9 \text{ bits/clock}$$

同样对于 $currwidth = 18 \text{ bits}$, $ch1$ 和 $ch2$ 的平均速率由式 7-18 进行计算:

$$avgrate(ch1) = (128 \times 23) \div [515 + (128 \times \left\lceil \frac{23}{18} \right\rceil \times 2)] = 2.86 \text{ bits/clock}$$

$$avgrate(ch2) = (128 \times 23) \div [129 + (128 \times \left\lceil \frac{23}{18} \right\rceil \times 2)] = 4.59 \text{ bits/clock}$$

由于两个通道的平均速率之和小于 $peakrate(B)$, 18 位的总线宽表示一个可行的实现。

我们现在来计算这个总线宽相应的代价。由于通道的峰值速率有一个 10 bits/clock 的最小约束, 其权重为 10, 而总线的峰值速率确定为 9 bits/clock, 所以

$$cost = (10 \times (10 - 9))^2 = 100$$

对 $<1, 23>$ 范围内总线宽度的其他取值重复上面的计算, 就产生一个总线宽和代价的对应图, 如图 8-7 所示。根据这个图, 最小的代价是 0, 对应的总线宽度是 20。该总线宽度将用来实现包含通道 $ch1$ 和 $ch2$ 的总线。

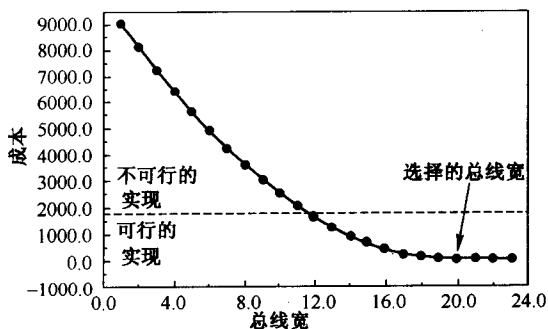


图 8-7 温度控制器: 成本与总线宽度的关系

图 8-8 显示出两个通过总线 B 传输数据的行为 $P1$ 和 $P2$ 的性能是如何被各种总线宽所影响的。对每个总线宽, 都采用一个性能评估器来获得行为的执行时间。很明显, 随着总线宽度的增加, 行为的执行时间减少。给定这些行为的性能约束, 设计者可以选择一个合适的宽度来实现总线。例如, 假设行为 $P2$ 有一个 2500 个时钟周期的最大执行时间约束, 如图 8-8 中

虚线所示。从这条虚线与 $P2$ 曲线的交点,我们可以得出结论:只有大于 4 bits/clock 的总线宽度才需要考虑用来实现总线。

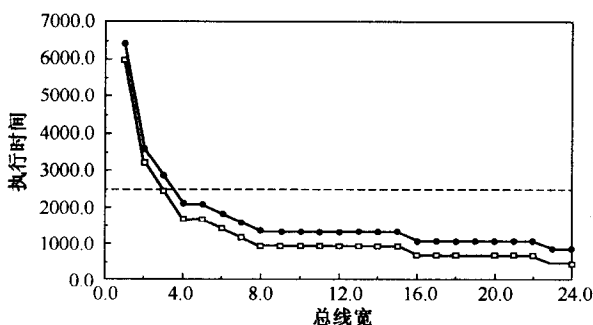


图 8-8 温度控制器: 执行时间与总线宽度的关系

8.3.4 协议生成

一旦我们选定了合适的总线宽度来实现通道组,协议生成工作就是要定义通过总线的数据传输机制。一条总线包括三组线:

数据线:用来通过总线传输数据。数据线的数量(总线宽度)可通过总线宽度生成算法来确定,或者由设计者规定。

控制线:用来同步通过总线进行通信的行为。控制线的数量取决于我们选择了哪种协议来实现数据传输。例如,标准的握手协议需要两个控制信号, $START$ 和 $DONE$ 。信号 $START$ 由主行为置位/复位,而 $DONE$ 由附于通道的从行为置位/复位。控制线是所有映射到同一条总线的通道共享的。

标志线或模式线:用来在任何时间点标记是哪一条通道在通过总线传输数据。由于总线控制信号是所有通道共享的,这种标志(identification, ID)线在本质上的重要性在于使行为辨别什么时候总线上的控制信号对其是有意义的。总线中的每个通道被赋予一个唯一的 ID,作为这个通道的地址。每次主行为通过总线发起一次数据传输,它就将相应通道的 ID 放到总线的 ID 线上,从而保证只有对应的从行为才对控制信号做出响应。ID 线也可以直接编码到通过总线访问的数据地址中。在这种情况下,从行为必须有一个地址检验机制,检查传送到总线的每个地址,以确定是否应该对主行为发送到总线上的控制信号做出响应。

我们通过一个简单的例子来说明协议生成,如图 8-9 所示。变量 X 和 MEM 被行为 P 和 Q 访问。虚线表示系统行为和变量到系统组件的分配。通道 $CH0$ 、 $CH1$ 、 $CH2$ 及 $CH3$ 被结群到总线 B ,而且总线宽度已经被确定为 8 位。协议生成包括如下几步:

1) **协议选择:**为实现一个总线可以选择各种通信协议,如完全握手协议,半握手协议,甚至硬件端口。每个协议需要不同数量的控制线。对于图 8-9 中的总线 B ,选择了完全握手协议。用两个控制信号 $START$ 和 $DONE$ 来实现握手协议。

2) **ID 分配:**如果 N 个通道实现在同一个总线上,则需要 $\log_2(N)$ 条线来对通道 ID 进行编码。每条通道被分配一个唯一的 ID。图 8-9 中的 4 条通道需要 2 条 ID 线。 $CH0$ 的 ID 是 00, $CH1$ 的 ID 是 01,以此类推。

3) **总线结构和过程定义:**为总线所确定的结构(数据、控制及 ID 线)是在描述中定义的。

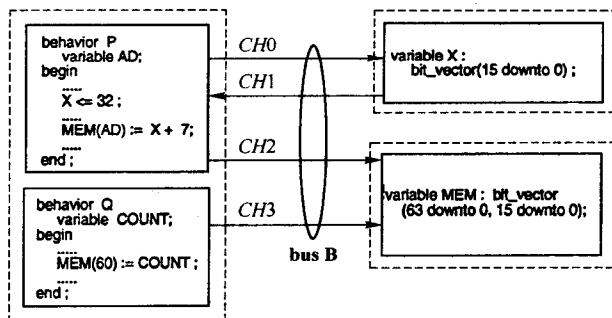


图 8-9 行为通过结群成总线的通道访问变量

```

type HandShakeBus is record
  START, DONE : bit;
  ID : bit_vector(1 downto 0);
  DATA : bit_vector(7 downto 0);
end record;

signal B : HandShakeBus;

procedure ReceiveCH0( rxdata : out bit_vector) is
begin
  for J in 1 to 2 loop
    wait until (B.START = '1') and (B.ID = "00");
    rxdata(8*J-1 downto 8*(J-1)) <= B.DATA;
    B.DONE <= '1';
    wait until (B.START = '0');
    B.DONE <= '0';
  end loop;
end ReceiveCH0;

procedure SendCH0( txdata : in bit_vector) is
  bus B.ID <= "00";
  for J in 1 to 2 loop
    B.DATA <= txdata(8*J-1 downto 8*(J-1));
    B.START <= '1';
    wait until (B.DONE = '1');
    B.START <= '0';
    wait until (B.DONE = '0');
  end loop;
end SendCH0;

```

图 8-10 定义总线 B 以及通道 CH0 的发送和接收协议

对于映射到总线上的每条通道,需要生成适当的发送和接收过程,封装对总线的控制线、数据线及 ID 线的一系列赋值操作以执行数据传输。图 8-10 显示出对一条 8 位总线的声明,它有两条控制线和两条 ID 线。总线 B 被声明为全局变量(在 VHDL 中就是一个信号),这样所有行为都可以访问它。行为 P 通过通道 CH0 向一个 16 位变量 X 写入。由于总线宽度只有 8 位,图 8-10 b 中通道 CH0 的过程 SendCH0 和 ReceiveCH0 通过总线发送一个 16 位的消息,分两次传递,每次传送 8 位。

327

4) **更新变量引用**:如果一个变量被系统划分过程分配到另一个系统组件上,那么原来对这个变量进行的直接引用就必须更新。对变量的访问将被替换成为发送和接收过程调用,该调用对应于进行变量访问的通道。例如,在图 8-9 中,行为 P 直接向 X 写了一个值“32”。CH0 现在负责向 X 写入数据。语句“X <= 32”被替换成为发送过程调用“sendCH0(32)”,如图 8-11 所示。行为 Q 中的语句“mem(60) := count”也被替换为“sendch3(60, count)”,指明 COUNT 的值应该写到数组 MEM 的地址 60 处。

328

5) **为变量生成进程**:为了获得可模拟的系统描述,为每组通过通道访问的变量都生成一个单独的行为。该行为包括合适的接收和发送过程调用,以响应通过总线对这些变量的访问

请求。在图 8-9 中,变量 X 和 MEM 被分配到不同的系统组件上,如虚线所示。在图 8-11 中,行为 $Xproc$ 和 $MEMproc$ 专门为这两个变量创建。

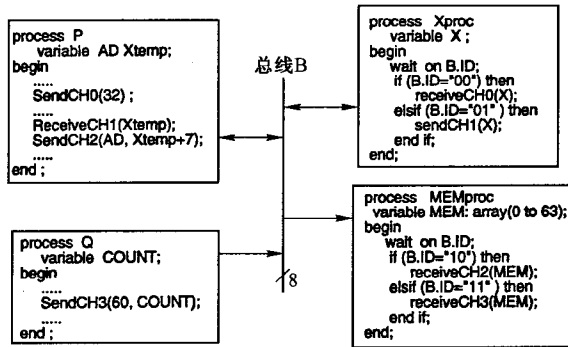


图 8-11 协议生成后的细化描述

本节介绍的协议生成有几点优势。首先,细化的描述是可模拟的,而设计的功能在插入了总线和通信协议后是可验证的。其次,通过对以发送和接收过程调用的数据传输的封装,即使我们在行为的每个通信点上都插入控制线 and 数据线,那么行为的描述依然清晰而不混乱。最后,即使在后面阶段选择了另一个通信协议,只有总线声明、发送过程、接收过程需要改变。系统行为描述,包括发送和接收过程的调用,都无需改变。

8.4 解决访问冲突

当系统中两个或多个行为试图同时访问同一个资源的时候,就会发生访问冲突。例如,系统划分中的变量和通道结群就会造成这种冲突。结群在一起的变量也许会被多个行为并发访问。如果实现一组变量的存储器的端口数比并发访问需要的少,那么访问冲突就会发生。类似地,通信通道也会被结群在一起,用一条总线实现。通过这些通道通信的行为也会试图同时进行数据传输,从而导致对总线的竞争。

仲裁(arbitration)用来解决由于多个行为并发访问一个共享资源而导致的冲突。仲裁生成是一个细化任务,用于当系统中存在资源竞争时,在描述中插入仲裁机制。

8.4.1 仲裁模型

资源通常拥有多个访问端口,支持有限数量的并发访问。这种资源的一个实例是多端口存储器,它有固定数量的端口。如果对这种资源进行并发访问的行为数量超过了访问端口数,我们就要解决并发访问冲突。通常采用两种仲裁模型:静态仲裁和动态仲裁。这里将以行为对一个 2 端口存储器进行访问的例子对这些模型加以说明,如图 8-12 所示。

在静态仲裁(static arbitration)模型中,一个行为的访问被分配到存储器的特定端口。在图 8-12a 中,行为 P 通过 $port2$ 访问存储器 Mem ,而行为 Q 和 R 通过 $port1$ 访问存储器 Mem 。访问到特定端口的映射是静态的,即在系统的生命周期中,一个行为都是通过分配给它的端口访问数据。在这种模型下,只有通过同一个端口的并发访问才被仲裁。因此,就需要仲裁器 $MemArbiter$ 来解决行为 Q 和 R 对 $port1$ 的访问冲突。虽然静态仲裁模型实现简单,但可能导致很差的性能。由于端口赋值是静态进行的,就可能需要一个行为等待静态分配给

它的端口可用,即使有其他端口空闲。

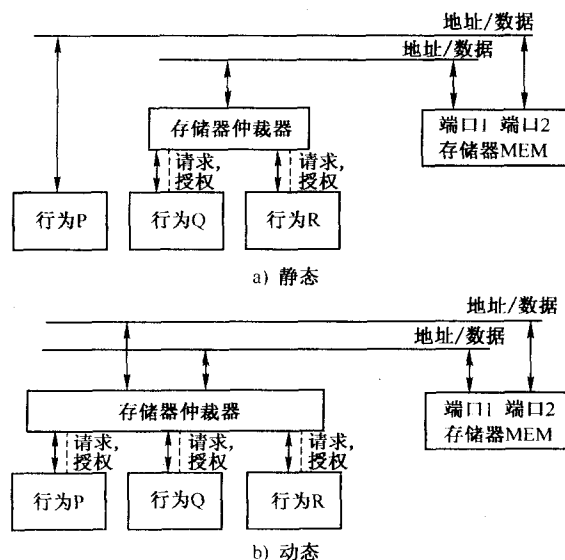


图 8-12 仲裁模型

在动态仲裁(dynamic arbitration)模型中,行为可以在不同的时间访问不同的端口,取决于端口是否可用。在图 8-12b 中,所有三个行为都通过开关网络连接到两个存储器端口。仲裁器 MemArbiter 限制在任何时刻最多只能有两个行为访问存储器 MEM。这种模型的主要优势是具有更高的端口利用率,并因此达到访问存储器行为的更快执行。然而,动态仲裁需要更复杂的实现,因为 MemArbiter 除了选择哪个行为被允许访问存储器以外,还必须把来自三个行为的数据连接到两个存储器端口。

331

8.4.2 仲裁方案

如上所述,多个行为访问共享资源的时候必须有先后顺序。给定一组行为,需要访问一组给定资源,仲裁方案就用于确定它们之间的相对优先级,以解决可能的访问冲突。仲裁方案可以分为固定优先级和动态优先级两种。我们下面分别描述。

1. 固定优先级

固定优先级(fixed-priority)方案为每个行为静态地分配一个优先级。行为间的相对优先级在整个系统的生命周期内都是不变的。如果两个行为同时请求访问一个共同的资源,则具有较高优先级的行为将被授予访问权。然而,静态优先级方案也有可能是抢占优先的,如果某个行为正在访问一个共享资源,而有一个更高优先级的行为也要访问,那么正在访问共享资源的行为就必须放弃访问而将访问转给高优先级的行为。一个抢占优先的例子就是 Intel 8237 的 DMA 控制器,允许较高优先级的外设抢占较低优先级外设的资源。

为各种行为确定固定优先级取决于度量,该度量还将被优化。其中一个度量是平均等待时间,表示任何行为等待对共享资源的访问所需的平均时间。一个低平均等待时间表示行为获取共享资源的速度很快。

这样,行为的优先级分配应该以最小化平均等待时间为目标。对于一个行为特定的优先

级分配,平均等待时间只能通过对描述进行模拟而动态地确定。为了静态地确定行为的优先级,平均等待时间通常用相对容易估计的度量来近似。一种度量就是行为访问的数据的大小。例如,如果某个行为通过通道一次性传输大量的数据,那么这个行为就会被赋予一个较低的优先级,以避免在数据传输期间,其他行为长时间都无法通过总线传递数据。另一个认为会降低平均等待时间的度量是行为对共享资源的访问频率(frequency of accesses)。行为的访问频率越高,优先级也就应该越高。

在确定行为的优先级的时候,系统设计者可以使用一种标准,也可以使用多种标准的加权组合。

2. 动态优先级

动态优先级(dynamic-priority)方案在系统运行的时候根据系统状态确定行为的优先级。例如,循环优先(round-robin)方案给最近访问共享资源的行为赋予最低的优先级。先来先服务(first-come-first-served)方案根据行为访问请求的到来顺序决定行为的优先级。这种方案的特点是对行为获得授权访问资源没有一个绝对顺序。所以,动态仲裁方案可看做平等的,换句话说,一个行为不会为了获得对共享资源的访问而无限等待。

8.4.3 仲裁器生成

一旦选择了仲裁方案,就以仲裁方案为基础为不同行为对共享资源的访问确定优先级。为了实现并发访问的仲裁,需要生成一个仲裁行为,并插入描述中。本节将以固定优先级仲裁方案为基础来说明为行为并发访问确定优先级的仲裁器的生成。

每个共享资源的仲裁机制都由一个独立的行为来表示,该行为和所有要访问该资源的行为都是并发的。在行为中,每次对共享资源的访问都被修改,以加入行为和仲裁器之间的握手机制。一个需要访问资源的行为将通过断言请求信号 *Req* 向仲裁器发出请求。仲裁器检查多个行为的访问请求,通过断言适当的 *Grant* 信号,给最高优先级的行为授予访问权。

算法 8.4.1:生成仲裁器

```
/* 在行为中引入请求和授权信号 */
for each behavior  $B_i$  which accesses resource  $R$  loop
    Precede all accesses of  $R$  in  $B_i$  by the following:
        Req_i <= '1';
        wait until (Grant_i = '1');
    Append the following after all accesses of  $R$  in  $B_i$ :
        Req_i <= '0';
end loop
/* 生成仲裁行为 */
Add the following to head of the arbiter behavior:
    wait untill Req_1 or Req_2 or ... or Req_N;
while priority_list  $\neq \phi$  loop
     $B_k = \text{First}(\text{priority\_list})$ 
    priority_list = Tail(priority_list)
    Append the following to the arbiter process:
        if (Req_k = '1') then
            Grant_k <= '1';
```

```

wait until (Req_k = '0');
Grant_k <= '0';
end if;
end for

```

算法 8.4.1[RVNG92]生成了一个给定资源 R 的仲裁器行为 VHDL 描述。 B_i 表示第 i 个访问资源 R 的行为。假设每个行为的优先级以降序保存在 $priority_list$ 中。过程 $First(L)$ 返回列表 L 的第一个元素,而过程 $Tail(L)$ 返回删除了 L 的第一个元素后的列表。

在图 8-11 中展示了行为 P 和 Q 的仲裁器生成, P 和 Q 通过总线 B 传输数据,总线的宽度和协议在 8.3 节中确定。假设行为 P 通过总线 B 传输数据的优先级比 Q 大。为总线 B 生成的仲裁器行为 $B_arbiter$ 如图 8-13 所示。适当的请求对总线访问的握手信号赋值已经插入到主行为 P 和 Q 中了。当行为 P 和 Q 需要通过总线传输数据的时候,它们分别通过断言 Req_P 和 Req_Q 来请求访问总线。行为 $B_arbiter$ 通过断言 $Grant_P$ 将总线访问权限赋予 P 。而 Q 只有在 P 不同时请求访问总线的时候才能得到访问总线的授权。

334

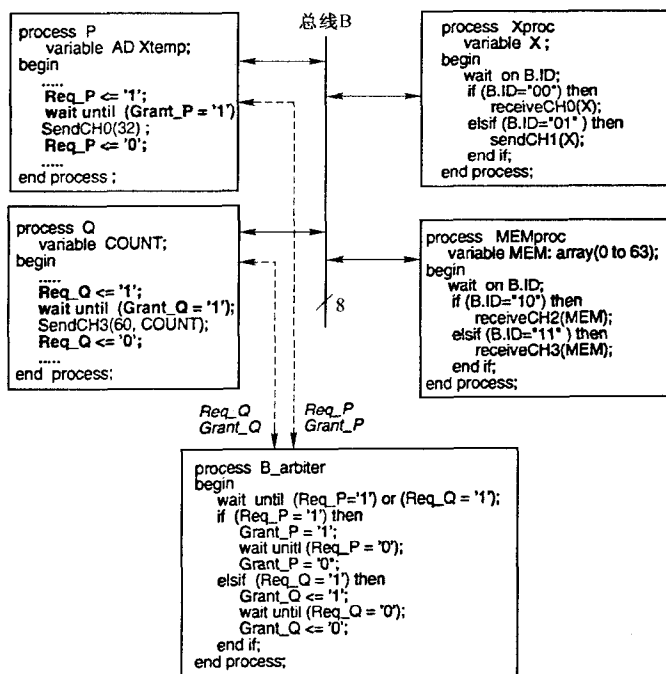


图 8-13 为总线 B 生成仲裁行为后的细化描述

8.5 细化不兼容接口

系统设计人员可能会把单一的或者一组功能对象绑定到一些成品组件上。变量组可能会被绑定到标准存储器中,行为可能会被绑定到标准芯片或处理器上,通道组可能会被绑定到标准总线协议上。这些标准组件的引脚结构和通信协议是固定的,不能改变。在系统组件之间通过不同的协议通信是可行的,但是必须引入合适的接口。

335

为了检验将功能对象绑定到标准系统组件的效果,我们来看图 8-14a 中的通道 X , A 和 B

通过它传输数据。假设 P_a 和 P_b 分别表示实现行为 A 和 B 通信的协议。根据哪些行为被绑定到标准组件,有以下 3 种情况:

1) 两个行为都没有被绑定在标准组件上,如图 8-14a 所示。它们之间的通信可以通过在 8.3 节所讲的方法,即生成总线并在行为 A 和 B 中插入协议来实现。

336

2) 其中一个行为被绑定在标准组件上,如图 8-14b 中的行为 B。协议 P_b 是这个标准组件对应的协议,是不可改变的。由于另一个行为 A 是定制设计的,它的协议被修改成为对于 P_b 是兼容的,用于保证行为 A 和 B 仍旧可能彼此通信。对 P_a 的修改会体现在行为 A 的描述中。

3) 两个行为都被绑定在标准组件上,如图 8-14c 中的行为 A 和 B。这种情况下, P_a 和 P_b 都是固定的。如果两个协议是兼容的,那么我们只需要简单地把两个标准组件的对应端口正确地连接起来就能确保它们能正确通信了。但是,如果 P_a 和 P_b 是不兼容的,两个标准组件之间就需要插入一个接口进程。接口进程见图 8-14c 中用虚线框起来的部分,它是一个行为,为两个协议不兼容的行为提供接口,以便于数据传输。

除了由于标准组件的使用导致的需求外,接口进程还用在其他的一些环境中。例如:由于引脚数量约束导致两个系统组件的数据引脚的数量不同。不同的数据引脚数量会导致两个系统组件的通信协议不兼容。这种情况下,就需要一个接口进程对两个协议进行连接。

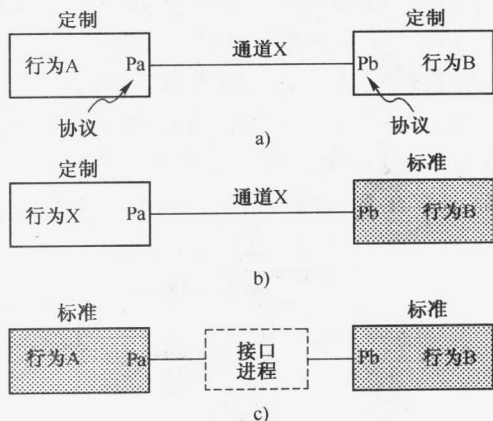


图 8-14 将行为绑定到定制或标准组件的影响

8.5.1 问题的定义

接口进程生成是一个细化任务,它在两个固定的且不兼容的协议的通信行为之间定义一个接口进程。接口进程生成的目标是生成一个接口进程,它正确地响应两个协议的控制信号,在它们之间传输数据。换句话说,接口进程把一个协议翻译给另一个协议。例如:一个组件的协议一次传输 16 位数据,但是另一个组件一次只能够接收 8 位数据。接口进程就从前一个组件接受 16 位数据,然后分两次,每次 8 位数据传递给后一个组件。接口进程生成的另一个目标是获得一个可模拟的细化描述。

337

8.5.2 通信协议描述

在阐述接口进程生成技术之前,我们简要考察一下通信协议描述的方法。

一个协议包含一组原子操作。原子操作有五类:(1) 在输入控制线等待一个事件;(2) 为输出控制线分配一个值;(3) 从输入数据线读取值;(4) 为输出数据线分配值;(5) 等待一个固定的时间间隔。任何协议都可以采用上面五类原子操作的组合来描述。

通信协议通常使用三种方式来描述:状态机、时序图和硬件描述语言。在图 8-15 中对每种描述都用例子示意。

在图 8-15a 中,行为 A 和 B 都被映射到标准组件。行为 A 读一个 $64K \times 16$ 的存储器,即行为 B 中的变量 $MemVar$ 。两个行为有固定的协议 P_a 和 P_b 。协议 P_a 有 8 位地址线、16 位数

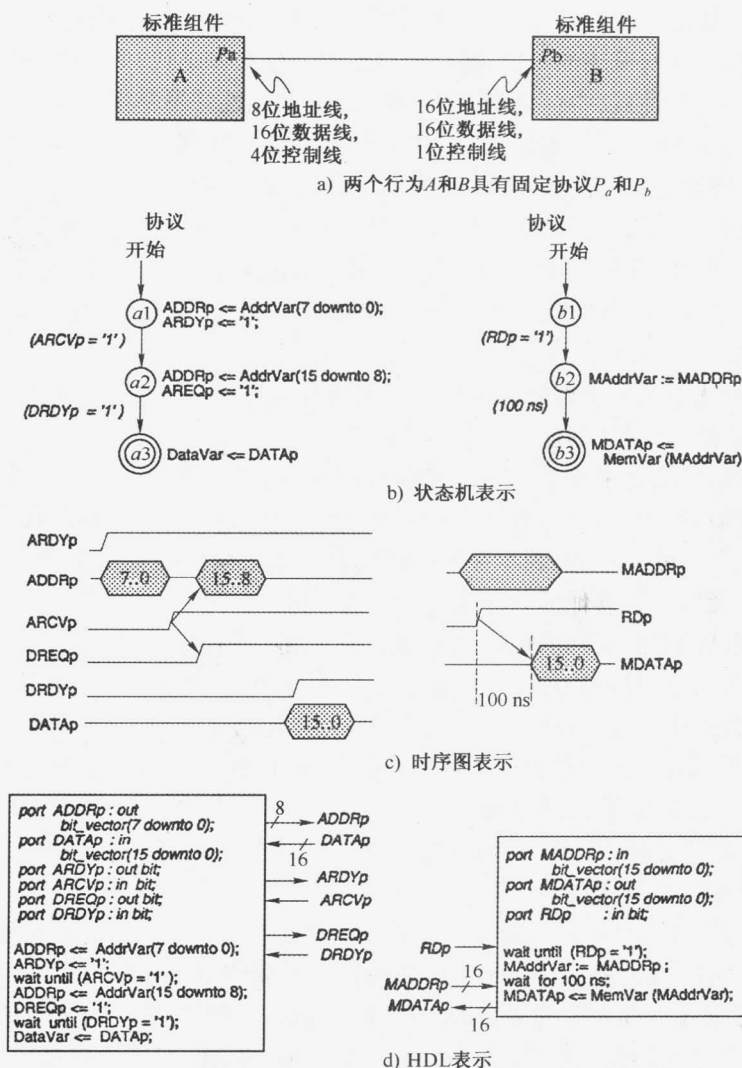


图 8-15 协议表示

据线和 4 位控制线,而协议 P_b 有 16 位地址线、16 位数据线和 1 位控制线。协议 P_a 使用的变量 $AddrVar$ 及 $DataVar$ 和协议 P_b 中使用的变量 $MemVar$ 都是相应行为的局部变量,提供分配到数据线的相关数据,或者接收从数据线读取的相关数据。例如,协议 P_a 中的变量 $AddrVar$ 提供了 16 位的地址,指向存储器中的一个单元,主行为 A 对其进行访问。为了便于识别,所有端口名称都有一个“ p ”后缀。

图 8-15b 显示了表示协议 P_a 和 P_b 的状态机,分别对应于行为 A 和 B。状态间的迁移由输入信号线上的事件的触发。例如:在 P_a 的状态机中, $AREQ_p$ 上升为“1”的事件,导致从 a_1 到 a_2 的状态迁移。在每个状态,都可以向端口赋值或者从端口读取值。在图 8-15b 中,状态 a_1 赋值到地址端口 $ADDR_p$ 和控制端口 $ARDY_p$ 。连续事件之间的时延可以直接描述为迁移的条件,如 P_b 状态机的状态 b_2 和 b_3 之间有 100 ns 的时延。其他方法还采用特殊的时延弧

(timing art)来表示协议中任何两个事件之间的时序约束。协议的状态机描述的缺点与在第3章中讨论概念模型特征时所概述的一样,需要明确地引入分支和迭代,因而需要额外增加状态及迁移。然而,由于大多数协议中包含的操作不多,所以局限性并不是很严重。

时序图(timing diagram)协议描述常见于半导体产品的数据手册中。在图 8-15c 中,显示了行为 A 和 B 中协议的时序图描述。协议的每个端口都在时序图中有一个对应的波形。从时序图中迁移的相对位置,可以推断出事件顺序。可在输入事件和输出赋值之间使用箭头来明确地指定顺序。例如,在协议 P_a 的时序图中,箭头确保了 $DREQ_p$ 在输入线 $ARCV_p$ 得到值“1”之后才有效。事件之间的时延可以如协议 P_b 的时序图所示来规定,在 RD_p 上升和 $DATA_p$ 数据输出之间有 100ns 的时延。

时序图最大的优点是易于理解协议操作发生的相对关系。另外,容许在每对事件之间直接表示时延约束。然而,时序图还有以下几个缺点。

首先,缺乏“动作”语言来表示与数据线相关的协议操作。换句话说,不能指定协议要传送数据的精确来源和目的地。例如,在图 8-15b 中,协议 P_b 的状态机表示读数组变量 $MemVar$ 中的一个元素。在时序图中只能表示在某些时候数据线上读来的数组变量的数据是有效的。同时,时序图也不能和系统其他部分的描述一起进行模拟。

其次,对重复性的事件序列的描述只能通过展开循环并在时序图上重复事件序列的办法,这将导致时序图过大。最后,采用时序图表示条件事件序列,需要多个时序图,协议中的每个条件或模式都需要一个。在[Bor88]中对上述一些问题进行了讨论,其中时序图被分解为一组带标记的段。将正则表达式语法应用于段标记的名称,使得能对重复和条件事件序列进行描述。在[MAP93]中提出的扩展时序图(ETD)容许将时序图分解成层次化的和并发的子图。在时序图的任何层次级别,都可以将动作附于事件和条件。通过从相应的 ETD 生成 VHDL 描述,协议行为就成为可模拟的。

图 8-15d 显示了用硬件描述语言(HDL)描述的两个协议。表示实际被传输的数据项的变量可在协议描述中指定,这就容许指定对诸如在任何时候变量的哪些位切片被分配到数据端口,以及哪些从数据端口读入等的细节进行描述。例如,在协议 P_a 中对 $ADDR_p$ 的两个赋值,指定了协议在两个不同的传输中,传输了地址 $AddrVar$ 的最低和最高有效字节。操作之间的时延用时延语句来规定(如 VHDL 中的 wait 语句)。

基于 HDL 的协议描述的主要优点是完备性——所有的端口信息(端口名、数目和类型),重复和条件协议操作都可以很容易地使用 HDL 的结构来表示,不需要额外的语言或注释。其次,协议可以与系统描述一起模拟,不仅验证数据传输,而且验证整个系统的功能正确性。基于 HDL 的协议描述的主要缺点是对事件间的时序约束进行描述非常麻烦。在 ISYN 系统[Nes87]中,将标号附于 ISPS[Bar81]接口描述的语句中。这些标号用于规定接口协议中相应操作之间的成对时序约束。

8.5.3 接口进程生成

我们现在就阐述生成接口进程的技术。接口进程生成的输入是两个固定协议的 HDL 描述,详细到控制线和数据线的数量,以及在这些线上数据传输的顺序。输出是接口进程的 HDL 描述,以及两个协议端口连接的相关信息。

我们将使用图 8-15a 中已经映射到标准组件上的行为 A 和 B 来阐明接口进程的生成。

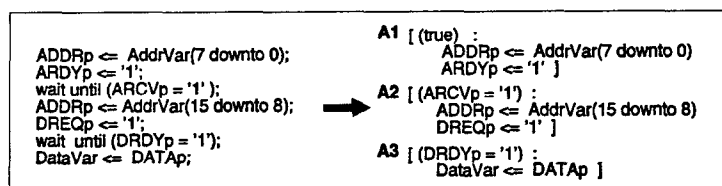
两个行为的协议是固定的 P_a 和 P_b , 其 HDL 描述在图 8-15d 中给出。

1. 将协议表示成有序关系

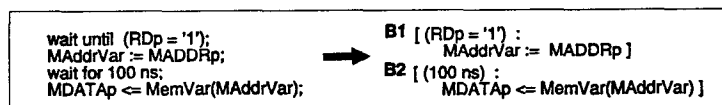
接口进程生成的第一步是将每两个协议表示成一组有序关系。关系定义了在某条件出现时, 一组对输出控制线和数据线的赋值, 以及对输入数据线的读入。条件可能是输入控制线上的一个事件或者相对于以前某个事件的固定时延。

图 8-16a 显示了如何从协议 P_a 的 HDL 描述中得到这组有序关系。前两个赋值语句没有任何先决条件。因此, 第一个标志为 A1 的关系, 包含默认为真的条件, 向端口“ADDR_p”和“ARDY_p”赋值。下一个语句 wait until (ARCV_p = '1') 表示在协议可以执行任何操作之前必须为真的条件。第二个标志为 A2 的关系, 就包含条件 (ARCV_p = '1'), 接着就向数据端口“ADDR_p”和控制端口“DREQ_p”赋值。最后, 第三个标志为 A3 的关系, 包含条件 (DRDY_p = '1'), 所以就将数据端口“DATA_p”读来的数据赋值给变量“DataVar”。

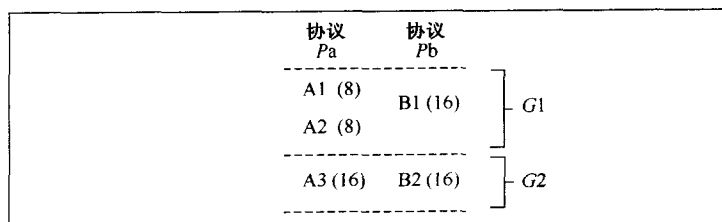
类似地, 可如图 8-16b 所示构造协议 P_b 的两个关系。第一个标志为 B1 的关系, 包含条件 (RD_p = '1'), 对内部变量 MAddrVar 赋值, MAddrVar 存储的是从端口 MADDR_p 读来的地址值。wait 语句规定的 100ns 时延表示后续操作的条件。因此, 关系 B2 包含时延条件 (100ns), 并接着向端口 MDATA_p 赋值。



a) 获得协议 P_a 的关系



b) 获得协议 P_b 的关系



c) 将两个协议的关系划分成关系组

图 8-16 关系的获得和划分

协议的关系化描述的优点是为捕获协议中事件之间 I/O 关系提供了一种简易的方式, 无论协议是如何描述的。在图 8-16a 和图 8-16b 中, 协议使用 HDL 来描述。但是协议也可以使用状态机或时序图来描述, 并同样可很容易地用一组关系来表示。

2. 将关系划分成块

当且仅当一个行为发送某特定大小的数据项, 同时另一个行为期待同样大小的数据项时, 才生成接口进程。例如, 在图 8-15a 中, 行为 A 发送 16 位地址, 同时行为 B 期待 16 位地址。

只要行为 A 发送的地址和行为 B 接收的地址的大小是一样的,即使 A 是每次发送 8 位地址并发送两次,而 B 一次就接收 16 位也无所谓。类似地,行为 B 向 A 发送从存储器中取出的 16 位的数据项,也与行为 A 所期待的数据大小一样。

得到两个协议的关系之后,需要将这两个协议的关系进行结群得到一系列的关系组。一个关系组是一组关系的一个有序子集,表示在两个行为之间进行数据传输的单位。创建关系组的方法是:一个协议中的组的关系所产生的数据大小应与另一个协议中的组的关系所产生的数据大小相等。

图 8-16c 显示出两个协议的关系是如何划分成关系组的。两个协议的关系列在表中,用括号括起来的数字表示在每个关系中操作传输的位数。协议 P_b 的关系 B1,从 $MADDR_b$ 读入 16 位数据。扫描 P_a 的关系列表,发现关系 A1 和 A2 都输出 16 位数据。这样,第一个组 G1 就包括关系 A1、A2 和 B1。归属两个协议的同一个关系组中关系的排序是由关系的数据依赖性决定的。只有在 A1、A2 产生地址之后,B1 才能读取这个 16 位的地址。所以在关系组 G1 中,A1、A2 在 B1 之前:

$$G1 = (A1A2B1)$$

关系组的记号规定一组关系 A1、A2 和 B1 的操作执行顺序为从左到右。

344

以类似的方式继续进行,我们通过合并关系 A3 和 B2 创建关系组 G2。由于关系 B2 的操作产生的 16 位数据,并被关系 A3 中的操作接收,所以在关系组 G2 中,关系 B2 在关系 A3 之前:

$$G2 = (B2A3)$$

3. 生成接口进程

将关系合并成关系组之后,就可以生成接口进程使得两个协议兼容。在关系组中按照顺序排列的操作集表示两个协议之间的原子操作的顺序。接口进程可以简单地通过转换关系组中的每个操作得到。“转换”一个原子操作意味着用对偶或者互补操作来替换它。

图 8-17 对 5 个原子协议操作给出了相应的对偶操作。例如,在一个输入控制端口等待一个事件, C_p 在接口进程中用它的对偶操作表示,即对控制信号 C_p 的一个赋值。向控制线 C_p 赋值的原子操作,在接口进程中有其对偶操作,该对偶操作就是等待相同的控制线来得到该赋值。协议向数据端口赋值表示为从数据端口读值到接口进程的本地变量中。协议从数据端口读值在接口进程中表示为从内部变量向数据端口赋值。

原子操作	HDL 等价表示	对偶操作
等待事件	wait until($C_p = '1'$)	$C_p \leqslant '1'$
控制线赋值	$C_p \leqslant '1'$	wait until($C_p = '1'$)
读数据线	$var \leqslant D_p$	$D_p \leqslant TempVar$
数据线赋值	$D_p \leqslant var$	$TempVar = D_p$
固定时延	wait for 100 ns	wait for 100 ns

图 8-17 原子协议操作的对偶操作

345

时延操作本身就是对偶的。为了了解什么时候需要将时延操作添加到接口进程中,考虑操作 o_1 和 o_2 : o_1 表示一个协议中的时延, o_2 表示在另一个协议中等待一个在输入控制线上的事件。如果同一个关系组中 o_2 在 o_1 后面,则时延操作 o_1 的对偶就包含在接口进程中。这就确保了另一个协议的操作 o_2 不会被过早地执行。例如,关系组 G_2 包含关系 B_2 和 A_3 。根据图 8-16 中对关系的定义,可以发现关系 B_2 中的条件(100 ns)之后是关系 A_3 中的等待条件($DRDY_P = '1'$)。为了确保协议

P_a 不会在协议 P_b 输出数据到数据线之前就读取数据线 $DATA_P$, 必须在接口进程中包含时延操作。

接口进程 IP 是通过在前面确定的关系组中, 对操作进行转换得到的:

$$\begin{aligned} IP &= (G'_1)(G'_2) \\ &= (A'_1 A'_2 B'_1)(B'_2 A'_3) \end{aligned}$$

在以上关系中, 将每个操作用相应的对偶操作代替, 我们就能够获得如图 8-18a 的接口进程。例如, 考虑关系 A_1 , 它包含了图 8-16a 中协议 P_a 的前两条语句的操作:

$ADDRp <= AddrVar(7 \text{ downto } 0);$

$ARDYp <= '1';$

A'_1 是这两个操作的对偶操作, 这就产生了如下语句:

$TempVar1(7 \text{ downto } 0) := ADDRp;$

$\text{wait until}(ARDYp = '1');$

这些语句是接口进程的前两个语句, 如图 8-18a 所示。

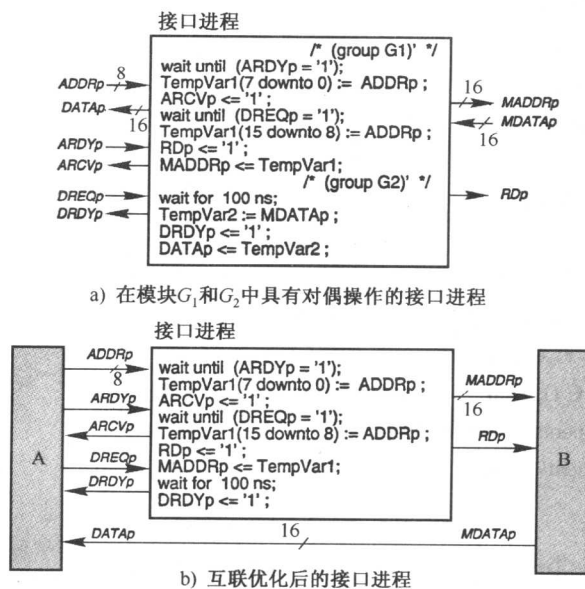


图 8-18 生成接口进程

接口进程需要的任何内部变量都在进程中声明。两个协议的每个控制线和数据线的端口声明都加入到接口进程中, 并且方向取反(协议的输入端口在接口进程中声明为输出端口, 反之亦然)。最后, 两个行为的控制和数据端口与接口进程中的相应端口连接。

4. 互联优化

前面的步骤生成的接口进程要求两个协议的数据线和控制线都连接到接口上, 如图 8-18a 所示。在一些情况中, 可将两个通信行为的某些控制和数据端口直接连接在一起, 有效地避开了接口进程。这样做有以下两个优点: 首先, 通过减少系统中的线网数量简化了系统的互联关系。其次, 在接口过程中和这些端口相关的操作可以完全删除。这样, 当进行逻辑综合时, 将产生一个更有效的接口进程, 面积更小并且性能更好。

第一种互联优化试图要做的是减少连接到接口进程的数据线。假设两个数据端口 D_1 和 D_2 , 大小相同并属于两个协议。如果接口进程每次从端口 D_2 读到数据都写到端口 D_1 , 并且在两个操作之间没有时延(wait 语句), 那么两个协议的数据端口就可以直接连接。所有向端口 D_1 的写入和从端口 D_2 的读出都可从接口进程中删除。因此, 临时生成的用于保存端口间传输值的变量就可以删除。在图 8-18a 中, 端口 $MDATA_p$ 和 $DATA_p$ 的大小都是 16 位, 从读 $MDATA_p$ 的数据到向 $DATA_p$ 写数据没有时延。这些端口就可以直接连接。此外, 临时变量 $TempVar2$ 和接口进程中对这些端口读写的语句都可以删除。

第二种优化检查两个协议的控制端口。考虑两个协议的控制端口 $C1$ 和 $C2$ 。如果每当接口进程从 $C1$ 得到一个特定的值, 就将 $C2$ 更新为这个值, 则 $C1$ 和 $C2$ 可以直接连接。例如, 在图 8-18a 中, 协议 P_a 和 P_b 的控制端口 $DREQ_p$ 和 RD_p 可以直接连接。相应的等待和赋值语句可从接口进程中删除。

为协议 P_a 和 P_b 生成的优化接口进程如图 8-18b 所示。数据端口 $MDATA_p$ 和 $DATA_p$, 控制端口 $DREQ_p$ 和 RD_p , 都直接连接。

算法 8.5.1: 生成接口进程

/* 为每个协议生成关系 */

$R_a = \text{CreateRelations}(P_a)$

$R_b = \text{CreateRelations}(P_b)$

/* 将关系划分成关系组 G */

$G = \text{GroupRelations}(R_a, R_b)$

/* 对每个关系组 G 中的操作, 在接口进程中加入其对偶操作的语句 */

for each relation group $G_i \in G$ **loop**

for each relation $R_j \in G_i$ **loop**

for each atomic operation $o_k \in R_j$ **loop**

$\text{AddDualStatement}(IP, o_k)$

end loop

end loop

end loop

$\text{CreateAndOptimizePorts}(IP, P_a, P_b)$

5. 评估接口进程生成

算法 8.5.1 总结了接口进程生成的步骤。给定协议的 HDL 描述, CreateRelations 生成表示该协议的关系集合。 GroupRelations 将集合 R_a 和 R_b 划分成关系组, 以集合 G 表示。对于关系组中按照顺序排列的每个原子操作, 过程 AddDualStatement 按照图 8-17 所定义的, 添加相应的对偶语句到接口进程 IP 中。一旦接口进程的语句生成之后, $\text{CreateAndOptimizePorts}$ 就在两个协议和接口进程之间生成一组端口, 如可能还进行优化。

本节介绍的技术的主要优点是可以在任何可用顺序 HDL 语句表示的两个协议之间创建接口。所生成的接口进程可与系统描述的其他部分一起进行模拟, 以验证在将一组功能对象

绑定到标准组件后系统的功能。该方法是通用的,因为它能为支持两个不同数据端口宽度的协议建立接口。该方法支持的时序信息表现为协议操作之间非重叠时延。但是,由于只生成了接口的 HDL 描述,不支持事件之间的最小和最大时延约束。这些约束可传递到综合工具,完成对接口进程的硬件综合。

8.5.4 协议兼容的其他方法

这里介绍解决标准组件之间的协议兼容问题的其他三种方法。

1. 转换器综合

定制芯片和系统总线之间的接口转换器的综合方法在[BK87, Bor88]中讨论。转换器(transducer)定义为连接两个电路块的粘连逻辑。两个不兼容接口的时序图作为输入。输出是转换器电路的逻辑描述。

我们以图 8-19 中 FIFO 栈控制单元[Bor88]为例,来说明如何从时序图对逻辑电路进行综合。该单元有两个输入和三个输出,如图 8-19a 所示。栈控制单元按异步操作,即没有外部时钟来同步单元内的操作。

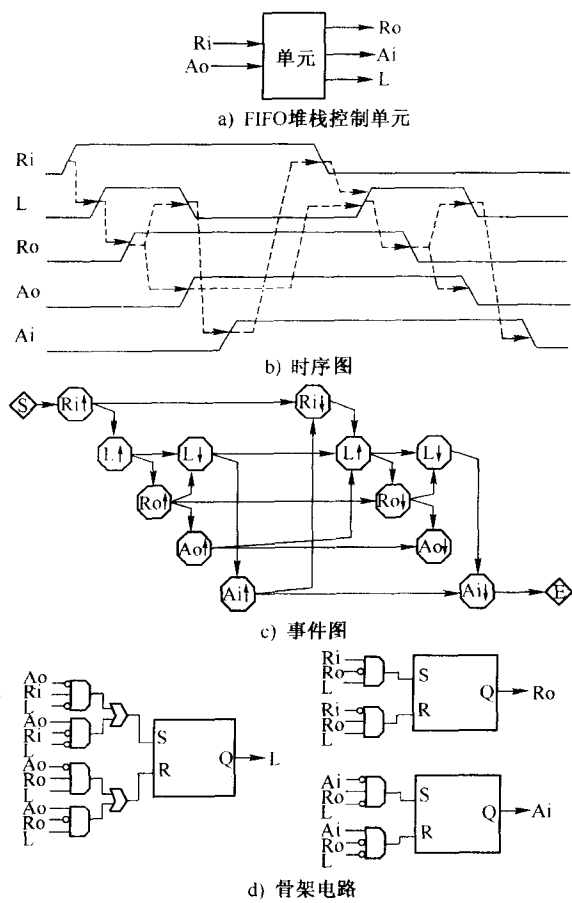


图 8-19 从时序图综合

图 8-19b 显示了单元输入和输出相关的时序图。虚线箭头表示单元中各种事件之间的有序

约束。第一步从时序图生成事件图。对于时序图中的每个事件,如上升和下降的跳变,都在事件图中对应一个结点。事件图中结点间的弧线表示时序图中相应事件之间的顺序或者最小和最大的时延约束。图 8-19c 显示出为图 8-19b 中时序图所构造的事件图。例如,对时序图中输出 L 的每个上升和下降沿的跳变,在事件图中就有相应的结点,分别标记为 $L \uparrow$ 和 $L \downarrow$ 。由于 L 的第一个上升沿是 R_i 的上升沿触发引起的,所以在事件图中从 $R_i \uparrow$ 到 $L \uparrow$ 加一条弧线。

一旦生成了事件图,就采用模板匹配策略来为单元产生的每个输出生成骨架电路。事件图中的每个结点都用一个模板电路实现,在输出产生正确的跳变。综合方法可采用三种不同类型的模板以反映要生成的各种事件:一种用于生成异步事件,一种用于从其他同步事件生成同步事件,一种用于从异步事件生成同步事件。每种模板都包含一个 SR 锁存器。上升和下降跳变是通过置位和复位 SR 锁存器来完成的。图 8-19d 显示出如何从图 8-19c 所示的事件图中为三个输出 L 、 R_o 、 A_i 中的每一个构造骨架电路。以单元输出 L 为例。 L 的第一个上升沿跳变是三个条件引发的:输入 A_o 为低,输入 R_i 为上升沿, L 为低。这样, L 的 SR 锁存器在 $\bar{A}_o R_i \bar{L}$ 为真时置位。这用一个与门来实现,该与门驱动锁存器的输入 S 以得到输出 L 。最后一步是应用逻辑优化技术使生成的骨架电路最小化。

在图 8-19 的例子中,逻辑电路是从单独的时序图得到的。为了综合转换器,首先从两个接口的时序图生成分开的事件图。接着将两个事件图合并成一个事件图,采用的方法是:或者明确地规定连接两个事件图中结点的合并标号,或者通过考察两个接口之间的数据依赖性。然后,将上面介绍的模板策略应用于合并后的事件图以生成骨架电路。接着,不同输出信号的锁存器通过对事件图进行宽度优先遍历的方法实现互联。通过添加适当的逻辑电路,可矫正任何违反时序约束和存在竞争条件等问题。例如,通过在电路中插入时延单元以满足最小时序约束。最后,对所产生的电路进行优化。

转换器综合方法的主要优势是结合了两个接口中事件之间详细的时序约束。其次,与前一节介绍的接口进程生成方法不同,转换器综合的输出是一个逻辑电路,也就不需要进一步的综合。该方法的一个局限性是不容许两个接口的数据宽度不匹配。此外,所产生的转换器电路不能和它所连接的协议时序图一起模拟。

2. 协议转换

协议转换器是一个行为,它使得两个协议的控制信号匹配,使得两者之间可以进行数据传输。协议转换器的综合方法在[AM 91, Ake91]中讨论。要建立接口的协议用基于 Verilog 的有限状态机来描述。将两个状态机进行叉积并优化就得到转换器的状态机描述。该方法可能导致协议转换器有非常多的状态。协议转换需要假设转换器中的数据通路是给定的,因此,与上面介绍的转换器综合方法一样,该方法也不支持数据宽度的不匹配。

3. 系统接口模块

在[SB92]中提出一种用于 SIERA 设计环境的系统接口模块的设计方法。SIERA [SSB91]通过提供一个包含详细 I/O 结构和协议(用事件图描述)的模块库,以寻求设计工作量的最小化。协议的细节在库中以事件图的形式表示,对系统设计者是不可见的。模块之间的通信被抽象为一个层次,在该层次上,设计者只需要从库中实例化适当的一些系统模块,用源端口和目标端口互联的方式规定这些模块的相互作用,采用专用语言 IDL 的高级元语(high-level primitive)来描述。

一个接口模块包含一个协议控制器和一个接口控制器。协议控制器对在单个传输中两个

协议的控制信号建立接口。接口控制器在两组数据线之间配置互联,并指导协议控制器对协议之间的握手事件建立接口。

首先,从用户对模块互联的描述中构造控制流图。将调度和分配应用于控制流图产生接口控制器,以及一个数据通道用以实现两个协议数据线之间的数据传输。其次,从模块库获得两个协议的事件图,并在基于两个协议中操作之间的数据依赖关系的基础上将两个事件图互联。从这个事件图中可综合出协议控制器,响应两个模块协议的控制信号。

这种方法的主要优势是将设计者从考虑底层细节的负担中解放出来,因为诸如 I/O 控制信号和时延约束等这些信息都被存储在模块库中。设计者只需要用 IDL 语言对高层次数据传输进行描述。虽然两个具有不同数据宽度的协议是可以接口的,但设计者需要明确规定两组数据线互联和多路选择,这些数据线要在接口模块中实现。该方法的另一个局限性是综合后的接口模块不能与原始协议一起模拟。

8.6 细化软件/硬件接口

任何行为描述都是既可以用软件实现也可以用硬件实现的。如果我们选择用软件实现,行为描述就被编译到选定处理器指令集;另一方面,如果我们决定用定制硬件来实现行为描述,就必须将描述综合成组件结构,其中的组件取自于给定库。这些组件可能是成品,也可用组件生成器生成。重要的是任何包含软件组件和硬件组件的系统都需要一个硬件-软件接口来实现不同部分之间的通信。

354

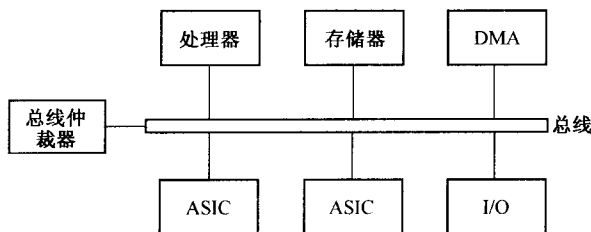


图 8-20 一种软硬件系统体系结构

在图 8-20 的系统中,处理器和存储器是软件组件,其他是硬件组件。这些硬件组件和软件组件用总线连接,这意味着每个组件要么是总线主端(bus master),要么是总线从端(bus slave)。总线主端是任何能控制总线、发起数据传输的组件,如:处理器、硬盘控制器、DMA 控制器。相对地,总线从端是任何能响应总线主端发出的命令,但不发起总线传输的组件。典型的总线从端是存储器和 I/O 组件。对于专用集成电路,按照其执行的特定功能,既可能是总线主端也可能是总线从端。例如,一个浮点协处理器一般作为总线从端实现,在协处理器中具有输入和输出缓冲器来存储操作数和结果。另一方面,一个用于捕获视频帧的专用集成电路一般实现为总线主端,因为这是获得高速存储器数据传输的最好方式。注意,当两个以上的总线主端连接到一个总线上时,我们就需要一个总线仲裁器来确保任何时候都只有一个总线主端控制总线。大多数总线,如 VMEbus 或者 Multibus,都在其协议中结合了总线仲裁方案。而另一些依赖处理器的总线,如 PC/XT/AT 等,都是由处理器对总线控制的并发请求进行仲裁。

总线自身作为一个互联组件,可为特定应用而设计,如 8.3.3 节中所述,或者也可以从预先确定好的总线集中选择,包括 Multibus、VMEbus、NuBus、PC/XT/AT bus、STDBus 等。由于

355

每种总线都有自己的通信协议,任何连接到总线的组件都必须遵守该总线的协议。如8.5节中介绍的,当某成品组件的通信协议与选定总线的协议不匹配时,我们就遇到了问题,此时为确保正确通信,设计者需要设计接口组件。注意,如果专用集成电路还没有实现,选定总线的协议可以集成在专用集成电路的功能中来解决接口问题。

下面我们将讨论若干与硬件组件和软件组件之间接口相关的一些任务。例如,变量分配过程可将变量分配到软件或硬件中,在最小成本下满足数据传输速率要求。接口生成的任务是添加一个接口进程,使得组件能与不兼容总线之间正确通信。最后,数据和控制访问细化的任务是将数据或者控制传输需要的通信协议插入到软件和硬件描述中。

8.6.1 目标体系结构

为解释软/硬件接口问题,我们以图8-20所示的体系结构为例。该体系结构非常简单,只包含一个处理器和几个专用集成电路。我们只使用一个处理器的原因是为了使软/硬件接口的细化更易于解释。更复杂的体系结构就需要考虑处理器间通信的同步和存储器保护的问题,这已超出了本书的范围。

通常,我们从一个划分描述开始软/硬件接口的细化工作,如图8-21所示。在这个描述中, $v1-v6$ 表示变量, $B1-B4$ 表示行为, $p1-p3$ 表示端口。图中的边表示行为对变量或者端口的数据访问。映射到软件部分的行为称为软件行为,类似地,映射到硬件部分的行为称为硬件行为。在本例中, $B1$ 和 $B2$ 是软件行为,而 $B3$ 和 $B4$ 是硬件行为。软件变量或软件端口定义为只能由软件行为访问的变量或端口。例如, $v1$ 是软件变量,

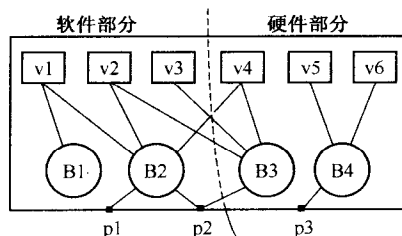


图 8-21 一个被划分的系统描述

$p1$ 是软件端口;类似地,只能由硬件行为访问的变量或端口称为硬件变量或硬件端口。在本例中, $v3, v5$ 和 $v6$ 是硬件变量, $p3$ 是硬件端口。最后,既能被软件行为访问也能被硬件行为访问的变量或端口称为共享变量或共享端口,如 $v2, v4$ 为共享变量, $p2$ 为共享端口。

8.6.2 变量分配

变量分配的过程就是将描述中的变量分配给软件或者硬件,也就是分配给存储器或者专用集成电路。通常,所有软件变量将分配给存储器,因为存储器一般足够大来容纳它们。我们也可以将软件变量分配给专用集成电路的存储器,但是没有任何好处,平白增大了专用集成电路成本。对于硬件变量,也倾向于分配给存储器,以适应专用集成电路芯片有限的尺寸,但我们必须认识到这种分配会增加总线通信量和降低专用集成电路的速度,其原因是专用集成电路可能会遭遇到与处理器的存储器访问竞争。当专用集成电路频繁访问这些变量的时,这种考虑尤其关键。我们还应意识到,任何共享变量是分配到存储器中还是分配到专用集成电路的存储器中,存在着类似于性能和成本的折中问题。

图8-22显示出三种可选择的变量分配方法。在这个例子中,行为A在一个处理器上执行,行为B1和B2被综合到专用集成电路中。需要注意的是行为A和B1都访问共享变量 $v1$,相应变量的通信通道分别是X和Y1。还需要注意的是变量 $v2$ 只通过通道Y2被B2访问。这三种变量分配方法的区别如下:图8-22a中的硬件变量和共享变量都分配给存储器;图8-22b中只有共享变量分配给存储器,而硬件变量分配给专用集成电路;图8-22c中变量

都分配给专用集成电路。

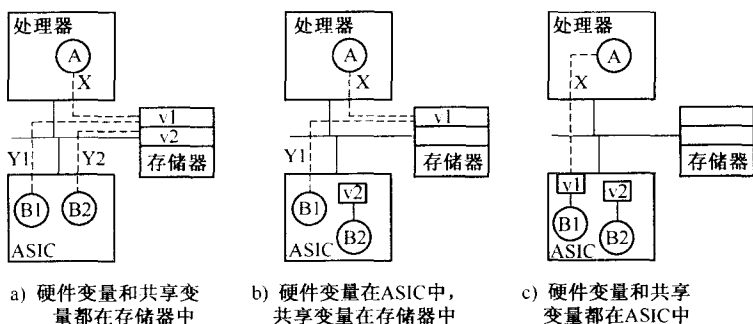


图 8-22 变量分配

假设通道 X 、 $Y1$ 和 $Y2$ 的平均速率分别为 $avgrate(X)$ 、 $avgrate(Y1)$ 和 $avgrate(Y2)$, 那么在图 8-22a、b 和 c 情况中, 所需总线传输速率分别为 $avgrate(X) + avgrate(Y1) + avgrate(Y2)$ 、 $avgrate(X) + avgrate(Y1)$ 和 $avgrate(X)$ 。对不同的变量分配, 我们可以确定其相对成本: 图 8-22b 计算为 $cost(v2)$, 图 8-22c 计算为 $cost(v1) + cost(v2)$, 其中 $cost(v)$ 表示在专用集成电路中实现变量 v 所需要的硅材料成本。注意, 由于 $v1$ 需要双端口, 所以 $cost(v1)$ 大于 $cost(v2)$ 。

358

从例子中可以看出, 降低总线传输速率的最好方法是将全部的硬件变量和共享变量都分配给专用集成电路。另一方面, 这样的变量分配将是昂贵的, 因为专用集成电路的成本远大于标准组件的成本。图 8-22 显示出需要把总线传输速率从情况 a 降低到情况 c, 付出额外的硅材料的成本。我们称这些额外硅材料的成本为变量-访问成本。

上面提到, 为了将软件和硬件建立接口, 我们必须选择一个总线以使得独立的软件组件和硬件组件可以通信。为了选择总线, 我们还必须考虑所需总线的传输速率和总线成本, 总线成本包括总线本身的成本及其接口组件的成本。如果组件协议与总线协议匹配, 那么接口成本为 0。另一方面, 如果能使用成品的接口电路来连接不相容的组件到总线, 那么接口成本就等于接口电路的成本。最后, 如果得不到这样的接口电路, 那么接口的成本将是很高的, 因为必须为接口综合出一个定制的专用集成电路。

我们应意识到, 总线选择和变量分配是互相关联的。例如, 对于一个给定的总线, 变量分配的目的就是寻找那种满足总线传输速率同时又使变量-访问成本最小化的分配方案。类似地, 对于一个给定的变量分配方案, 总线选择的过程就涉及选取满足所要求传输速率且成本最小的总线。在大多数情况下, 由于我们需要以最低系统成本满足数据传输速率的一个解决方案, 因此要统筹考虑总线选择和变量分配。

在算法 8.6.1 给出一个可用于并发总线选择和变量分配的算法。直观来看, 算法检查一组可能的总线 S_b , S_b 可由设计者提供, 或者存于一个包含了所有可用总线的库中。对每个属于 S_b 的总线 B , 我们要对一组变量 S_v 考虑每种可能的变量分配 D , S_v 包含的要么是划分后系统描述中给出的所有的硬件变量和共享变量, 要么只是设计者在这些变量中选出的一个子集。注意, S_v 不包含软件变量, 因为所有软件变量应自动分配给存储器。

359

算法 8.6.1: 变量分配

Determine S_b /* 总线集合 */

Determine S_v /* 变量集合 */

$mincost = \infty$

$mincost_bus = unknown$

$mincost_var_dist = unknown$

for each $B \in S_b$ **loop**

for each D of S_v **loop**

$datarate(D, B) = \sum_{C \text{ mapped to } B} avgrate(C)$

if ($datarate(D, B) \leq rate(B)$) **then**

$currcost = Cost(B) + Cost(D)$

if ($currcost < mincost$) **then**

$mincost = currcost$

$mincost_bus = B$

$mincost_var_dist = D$

end if

end if

end loop

end loop

if ($mincost = \infty$)

then **return** (*failure*)

else **return** ($mincost_bus, mincost_var_dist$)

end if

360

对每种变量分配 D , 我们通过计算映射到总线 B 的所有通道 C 的平均速率 $avgrate(C)$ 的总和 ($avgrate(C)$ 的定义见 7.2.3 节), 就能确定所选总线 B 上所需的数据传输速率, 表示为 $datarate(D, B)$ 。这样, 我们就能确定给定总线 B 的传输速率 $rate(B)$ 是否满足特定配置所需的数据传输速率。该算法会返回满足所需数据传输速率并且成本最小的配置方案。成本包括总线成本 $Cost(B)$ 和变量 - 访问成本 $Cost(D)$ 。注意: 临时变量 $mincost$ 、 $mincost_bus$ 、 $mincost_var_dist$ 被用来记录迄今找到的最优配置方案。

算法复杂度为 $O(N \times M)$, 其中 N 是总线数量, M 是要检查的变量分配方案数量。尤其需要注意, 由于 $rate(B)$ 通常作为总线 B 的峰值速率给出, 算法应将 $datarate(D, B)$ 计算为所有通道 C 的峰值速率的总和 $peakrate(C)$, C 为映射到总线 B 的所有通道 ($peakrate(C)$ 的定义见 7.2.3 节)。然而, 在很多情况下, 采用通道峰值速率计算过于保守, 可能导致对总线的低效使用, 因为在大部分时间, 通道不大可能以峰值速率传输数据。要选择 $peakrate(C)$ 而不是 $avgrate(C)$ 来计算 $datarate(D, B)$, 就需要对数据传输情况进行剖析。

8.6.3 接口生成

如前所述, 设计者有时候遇到这样一种情况, 即所选总线和所选组件的通信协议之间不兼容, 并且没有成品的接口组件可以使用。在这些情况下, 需要使用一个类似于 8.5 节中讨论的进程生成一个接口, 用以进行协议转换。在 8.5 节中, 我们提出了一种在两个不兼容组件之间

生成接口的技术,包含以下步骤:(1) 将要建立接口的组件协议表示成有序关系;(2) 将这些关系划分成关系组;(3) 通过将关系组中的每个操作转化为对偶操作来生成接口的描述。

361

在组件和总线之间生成接口的技术与两个组件之间生成接口的技术相似,只有一个微小差别。我们使用图 8-23 的例子来说明这个差别。假设图 8-23a 中的组件 A 使用它的协议向组件 B 发送数据,组件 B 使用自己的协议接收数据。两者之间的接口为了适应 A 和 B 的协议的不兼容性,按照与 A 匹配的协议接收来自 A 数据,然后按照与 B 匹配的协议向 B 发送数据。如在 8.5.3 节中介绍的,这些在接口中匹配的协议将由与 A 和 B 中操作对偶的操作

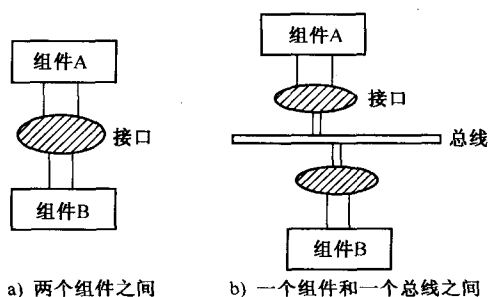


图 8-23 接口

组成。在图 8-23b 中,我们需要的是组件 A 和总线之间的接口,该接口按照与组件 A 匹配的协议接收来自 A 的数据,然后与总线相同的协议将数据发送到总线上。类似地,我们在总线和组件 B 之间也需要一个接口,该接口按照总线的协议接收数据,然后按照与组件 B 匹配的协议向 B 发送数据。需要注意的是,如果需要一个接口,使用与某组件匹配的协议,该接口就包含与组件协议操作对偶的操作。另一方面,如果需要一个接口,使用与某组件相同的协议,该接口就包含与组件协议操作完全相同的操作。因此,总线和组件之间的接口要包含那些与组件的协议操作对偶的操作,也要包含与总线协议操作相同的操作。

362

图 8-24 是一个接口实例,该接口实现了组件连接到总线的组件的读操作,该总线是 VME 总线的简化版本。可以看到如图 8-24d 中所示的接口,包含了组件协议操作的对偶操作,以及与总线协议操作相同的操作(着重显示部分)。注意信号 AS 的星号表示 AS 是低电平有效的。协议生成的详细内容已经在 8.5.3 节中讨论到了。

8.6.4 数据访问细化

当我们将行为和它访问的变量分配到不同划分部分的时候,由于变量定义已被从行为中移动出来,我们就需要对描述中的变量访问进行细化。返回到图 8-21 中的描述,假设在变量分配之后,共享变量 v_2 和硬件变量 v_5 被分配给存储器,而共享变量 v_4 被分配给专用集成电路的双端口缓冲器,如图 8-25 所示。在同一个划分部分中的数据访问并不是问题,因为它们或者如图 8-25 中的 v_1 那样由软件编译照顾到,或者如 v_3 和 v_6 那样由硬件综合工具来处理。因此,数据访问细化的任务是要对跨越不同划分部分的数据访问进行细化,如: v_2 、 v_4 和 v_5 的情况。

在图 8-25 中,存储器或专用集成电路缓冲器中的每个位置都在全局地址空间中有一个唯一的地址,该地址对任何总线主端都是可见的。由于通常处理器的引脚数是固定的,没有额外引脚供端口使用,所以软件行为访问的端口必须映射到全局地址空间中,并通过处理器的总线访问。所以,软件行为的端口访问也需要细化。注意,硬件行为的端口访问不需要细化,这是由于专用集成电路可为端口提供引脚。在图 8-25 中,虚线框中部分的数据访问通道都需要细化。

363

364

数据访问有四种基本类型:

1) **软件行为访问存储器**:该类型的数据访问通过处理器的取/存指令完成。在图 8-25 中, B2 对 v_2 的访问属于这种类型。

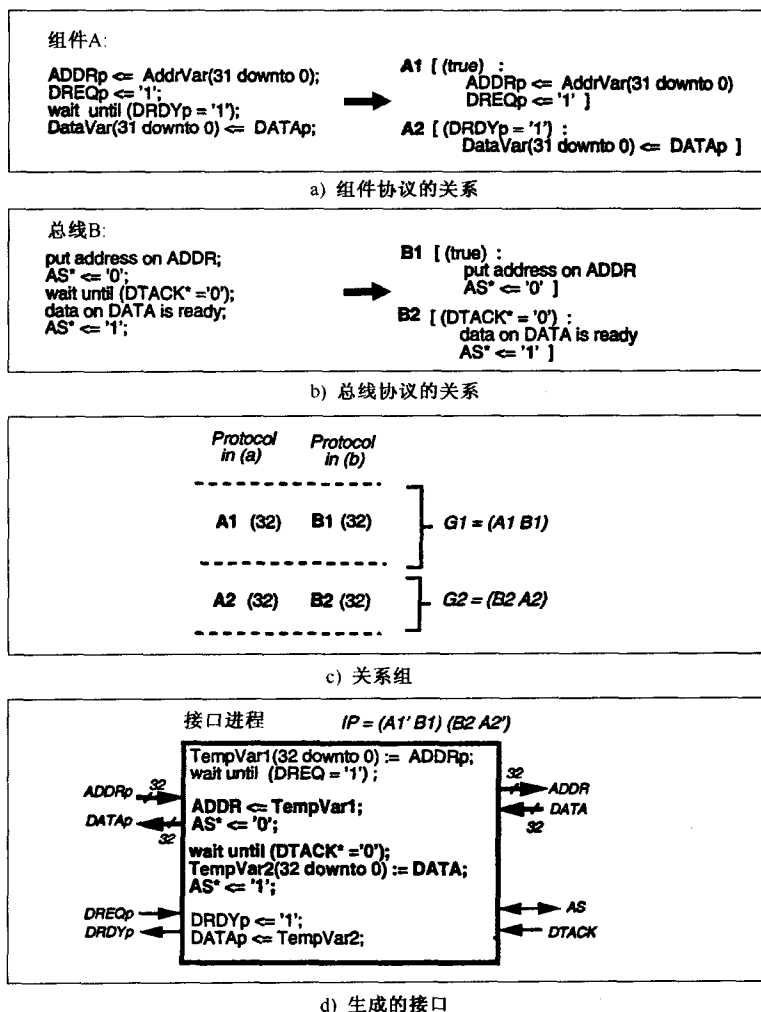


图 8-24 组件-总线接口

2) 硬件行为访问存储器: 由于该类型的数据访问是专用集成电路通过直接存储器存取 (DMA) 机制完成的, 所以我们称之为 DMA 操作。这种访问基本与取和存操作类似, 不同的是专用集成电路要先得到总线的控制权。在图 8-25 中, B3 对 v_2 , B4 对 v_5 的访问属于这种类型。

3) 软件行为访问端口或者专用集成电路的缓冲器: 该类型的数据访问通过处理器的输入/输出或者转移指令完成。在图 8-25 中, B2 对 p_1 , p_2 , v_4 的访问就属于这种类型。

4) 硬件行为访问专用集成电路的缓冲器: 由于该类型的数据访问是专用集成电路对其缓冲器访问完成的, 所以我们称之为缓冲器操作。在专用集成电路中, 这些操作使用独立的总线。在图 8-25 中, B3 对 v_4 的访问属于这种类型。

数据访问细化的任务规定如下:

1) 为已经映射到全局地址空间的变量和端口分配地址。

2) 用取和存操作代替软件行为对这些已分配到存储器的变量的访问。类似地, 用 DMA 操作代替硬件行为对这些变量的访问。

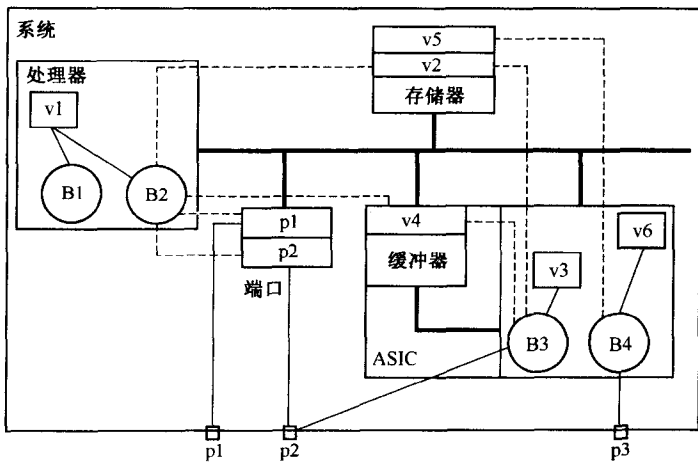


图 8-25 系统描述映射到目标体系结构

3) 用输入/输出操作代替软件行为对这些已映射到专用集成电路的变量的访问。用缓冲器操作代替硬件行为对这些变量的访问。

4) 用输入/输出操作代替任何软件行为对映射到全局地址空间的端口的访问。

8.6.5 控制访问细化

不同于行为和变量之间的数据访问通道,控制通道存在于两个行为之间用来指示行为的开始或者完成。图 8-26a 显示了对行为 B_1 、 B_2 、 B_3 顺序调度的例子,即 B_1 结束之后 B_2 开始, B_2 结束之后 B_3 开始。这样在 B_1 与 B_2 、 B_2 与 B_3 之间存在控制通道。如果行为 B_2 由硬件实现,则连接到 B_2 的控制通道就必须进行细化,使之和硬件/软件接口一致。

一个解决方案是引入握手协议,使用两个共享变量: $start$ 和 $done$ 。协议语句会被插入到 B_2 位置, B_2 本身会被加入匹配协议语句,如图 8-26b 所示。如果我们将变量 $start$ 和 $done$ 映射到双端口缓冲器,图 8-26b 所示的描述可被映射到图 8-26c 所示的目标体系结构。控制细化可以在对共享变量 $start$ 和 $done$ 的数据访问细化完成后进行。

366

由于 B_2 移到硬件实现并不影响其余的软件部分,所以这个控制细化方案是简单的。但是我们还应意识到,这个方案需要处理器对存储变量 $done$ 值的地址进行轮询,这可能造成处理器时钟周期的浪费。

由于商用处理器通常提供中断机制,所以可能采用更有效的控制细化方法。例如,在图 8-27a 中,我们展示了一种采用中断机制来指示硬件行为完成的方案。在这个例子中, B_2 被一个新的行为取代,该行为包括一个开始 B_2 的协议语句。而且,行为 B_2 已经被扩展,前端包含协议语句,后端包含中断语句。对于每个中断,在软件划分部分中插入相应的服务例程。当中断发生时,控制权会被转移给服务例程。使用中断机制,处理器可以在 B_2 完成之前就开始 B_3 ,只要 B_2 和 B_3 之间不存在数据相关。当然,处理器也可以选择开始另一个软件行为,如 B_4 。

367

控制访问细化的任务总结如下:

368

1) 选择一个控制方案,如轮询或者中断。然后在软件和硬件行为中插入相应的通信协议。

2) 插入必要的软件行为,如中断服务例程。

3) 对协议引入的任何共享变量的访问进行细化。

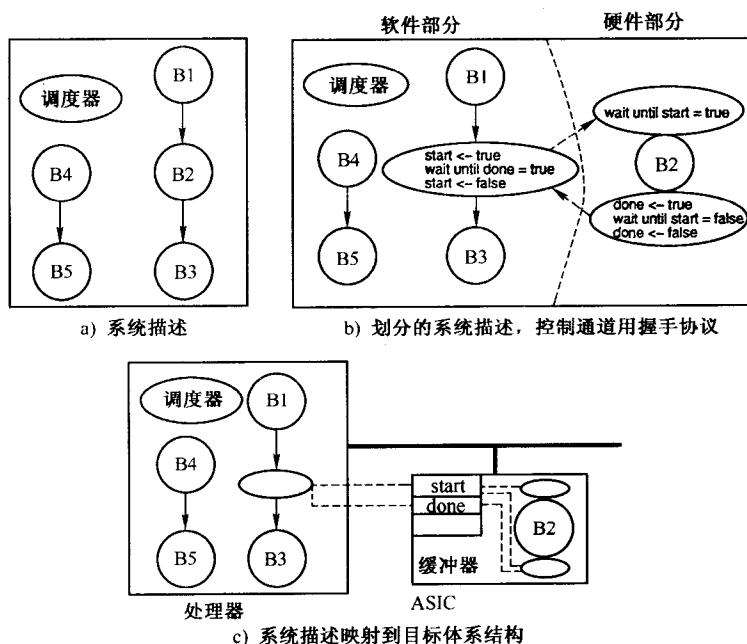


图 8-26 控制访问细化

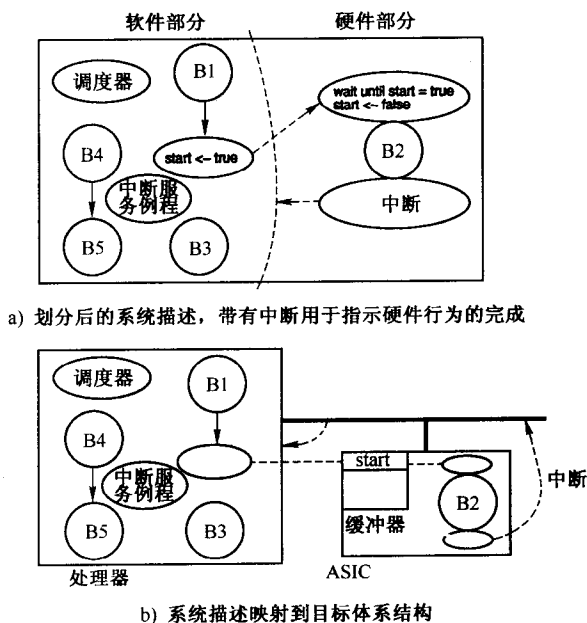


图 8-27 控制访问细化

8.7 结论和发展方向

在本章中,我们提出了在系统划分之后必须执行的一系列细化系统描述的任务。我们首

先论证了在系统设计中进行系统描述细化的重要性。我们还讨论了与变量和通道组实现相关的问题。还介绍了一种总线生成的方法,并评估了总线宽度和系统性能之间的折中。对于选定的总线宽度,我们展示了如何生成通信协议。还讨论了对系统中共享资源的访问冲突的解决方法。我们还考察了绑定功能对象到成品组件对通信的影响。我们描述了对两个固定和不兼容的协议建立接口的技术。最后,我们提出了行为之间通信的细化方法,这些行为已映射到硬件和软件来实现。

本章介绍的工作可以在以下几个方面进行扩展。首先,在协议生成过程中,需要制定度量用以对选择来实现总线传输的若干种候选协议进行评估。其次,在我们讨论总线生成的过程中,仲裁时延假定是可忽略的,但并非总是如此。我们需要在行为执行时间和通道平均速率中结合进仲裁时延的影响。再次,不同总线仲裁方案对在总线上进行通信的行为的性能影响也要进行研究。

需要研究的还有优化,可应用于使两个协议兼容的接口进程。这种优化的一个例子就是最小化接口进程所需变量的数量和大小,以减小实现接口进程的硬件大小。最后,虽然硬件描述语言对仲裁器和接口进程的描述容许设计者对其连同系统描述一起来进行模拟,但采用传统的基于硬件描述语言的综合方法对这些进程进行综合将导致低效的设计。例如,对仲裁进程进行调度这样的综合任务将导致仲裁器的多状态实现。然而,仲裁器通常只包含组合逻辑。这样,就需要开发方法以有效地从硬件描述语言的描述对接口和仲裁进程进行综合。

369

8.8 练习

1. 一个变量组用具有 16 位字宽的存储器实现。在存储器的同一个地址映射两个 8 位宽的变量,一个放在高 8 位,一个放在低 8 位。讨论这种实现方式的优缺点。
2. 如果一个包含 100 个元素的数组 A 被分配到地址从 201 到 300 的存储器 MEM 中,在下面的一组语句中更新对 A 的引用:


```

X := A(30);
for I in 30 to 60 loop
    Y := A(I + 20) * A(I - 20) + Y;
end for;
A(X + A(10)) := 3;

```
3. 假设一个行为 B 通过一组通道和其他行为通信。如何修改计算通道 C 的平均速率的公式 7-18,使之将其他通道所需的通信时间考虑进来。
4. 在图 8-10 中,对于一个通过 8 位总线访问的标量变量生成了发送和接收过程。请生成用以通过总线访问一个数组变量 A 的发送、接收过程。假设 A 有 256 个元素,每个元素有 16 位宽。
5. 假设对某共享资源有 N 个并发访问。请生成一个实现循环优先级仲裁方案的仲裁进程。
6. 对于图 8-12b 中的动态仲裁模型,概述 $MemArbiter$ 的互联逻辑。请清楚地描述行为 A 、 B 和 C 的地址/数据线是如何和两个存储器端口的地址/数据线连接的。假设每个行为都可对任一存储器端口进行读或写。
7. N 个通道并入一个总线。对这样的总线进行仲裁,每个通道需要两个信号: $request$ 和 $grant$,即总共有 $2N$ 个信号。设计一个使仲裁信号数量最小化的算法,方法是在使用时间

370

上互斥的通道间共享仲裁信号。

8. 假设一组访问共享资源的行为具有在执行时间上的约束。如果采用固定优先级的方案来解决访问冲突,如何确定行为的相对优先级。
9. 为算法 8.6.1 的变量分配设计启发式方法以修剪搜索空间。
10. 应用 8.6.3 节中介绍的技术生成一个接口,用以将 Intel 8086 的数据传输操作连接到 VME 总线。
11. 请举例说明行为执行的顺序如何会影响 I/O 数据速率。设计重新调度软件行为或者操作的技术,使得能满足 I/O 数据速率约束。
- ** 12. 一组行为,通过一组通道通信,将用单总线实现。设计一种策略,能预测仲裁时延对行为执行时间的影响。

第9章 系统设计方法学

在本书的前几章里,我们介绍了系统描述和设计中的一些核心问题和技术。为了尽可能利用这些技术,需要将它们结合进一个能易于应用到实际系统的、具有一致性和包容性的方法学中。在这一章里,我们将提出这样一种系统设计方法学,概述能为该方法学提供最好支持的工具,并描述这些工具和这种方法学如何与已有的设计实践相联系。

9.1 引言

每个产品从最初的概念化到最后制造,都需要经过许多设计阶段和任务。这个过程叫做设计过程,而这一系列的设计任务和相关的 CAD 工具的技术被称为设计方法学。在较低的抽象级别,设计方法学有清晰的定义,并得到了较多商业 CAD 工具的支持。然而在系统级,这种定义并不是特别清晰,并且可用的 CAD 工具也很少。此外,不同组织、设计组及产品的系统方法学都有所不同。在这一章里,我们将提出一种具有一致性的系统级方法学,来帮助学术界和商业界研究从逻辑和结构级到系统级之间的模式转换。

373

在前面几章里,我们定义了系统级设计,对系统描述的模型和语言进行了讨论,并提出了有关划分、评估和细化方面的问题和算法。在这一章里,我们将把这些概念联系起来,形成系统级方法学。首先,我们通过一个例子来解释方法学。然后,结合所提出的方法学,我们描述了一个假想的设计过程及相应的综合系统。我们还将简单介绍在各个抽象级别上的设计工具。最后,我们对支持系统级工具的环境进行讨论。

9.2 基本概念

在提出系统级方法之前,我们首先对设计方法学的内涵进行定义。一个设计方法学必须清楚地指明以下几点:

- (a) 输入和输出描述的语法和语义;
- (b) 将输入转换为输出描述的技术集合;
- (c) 设计实现中用到的组件集合;
- (d) 设计约束的定义和范围;
- (e) 组件和体系结构的选择机制;
- (f) 设计探索策略(通常称为剧本或脚本),用于定义综合任务及其参数,以及执行顺序。

通常(a)和(b)可以通过对描述语言和综合工具集合的选择来定义,而(c)和(d)则可以由选择的工艺和系统体系结构来确定。尚未定义的(e)和(f)则是设计者的任务。下面将通过一个例子来解释上述这些从(a)到(f)的需求,并得出一个满足这些需求的综合系统。

374

9.3 设计方法学举例

这里考虑一个交互式电视处理器(ITVP)的例子。在播放音频的同时,系统把存储的视频帧当作静止图片显示出来。用户可以通过键盘、遥控器或触摸屏来选择菜单项,结果是显示出新的视频帧,并伴有音频。这种系统在旅馆、商店和邮局里很常见,将来还可能在交互式电视多媒体

环境里使用。这些系统可用作旅馆的视频导航、通过视频目录来购物、玩视频游戏、处理银行事务或者预订航班。该系统置于监视器或电视机附近的一个盒子里,就像一个有线电视机盒。整个系统示意图如图 9-1 所示,其中只给出了部分数据流。一个模拟子系统将模拟信号转换为数字信号,并从视频输入中提取出各种同步信号。另一个模拟子系统将数字信号转换为模拟信号,同时也执行其他各种任务。ITVP 的核心是一个数字子系统,这就是我们设计的对象。

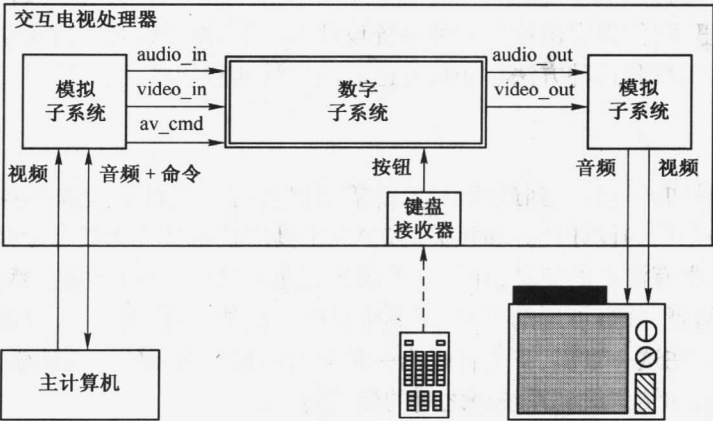


图 9-1 ITVP 的环境

375

对于图 9-1 中的数字子系统,其主要行为、数据对象及数据流如图 9-2 所示。实际系统中含有 32 个行为和 69 个数据对象,但为了简化,图中只显示出较大和重要的行为和对象。系统中的行为 *StoreAudio* 通过调制解调器从音频输入接受大量连续音频字节并进行存储;行为 *GenerateAudio* 根据命令来产生这些音频字节。由于系统可能在产生音频字节的同时也在存储音频字节,因此需要用两个数组 *audio1* 和 *audio2* 来存储这些音频字节。行为 *StoreGenerateVideo* 的功能是存储和产生视频帧,它与一个用于存储视频字节的数组 *video* 一起发生作用。对于系统支持的 128 个 ASCII 字符,数组 *fonts* 指明在一个 16×16 的点阵中,应该使哪些像素明亮,以显示任何一个字符。另一个数组 *screen_chars* 指明在 30×30 个屏幕位置上,任何位置显示哪个字符。行为 *OverlayCharacters* 读取屏幕字符和字体(font)数组,据此向视频生成器指明何时需要用白色的像素来覆盖视频像素,使得白字符可以在屏幕上显示。行为 *StoreAVCmd* 和变量 *av_cmd* 获取一个被编码的命令,指明那个音频数组用来存储接受的音频字节,以及从哪个数组产生输出。这个命令还规定了视频输入的数据类型,使得系统可以扩展来用软件编码视频。软件将在处理器上运行,来处理时间约束更严格的交互式程序,如视频游戏。行为 *ProcessRemoteButtons* 对键盘或遥控器上的按钮动作做出反应。行为 *ProcessMainCmds* 对主计算机发出的命令做出响应,而行为 *ProcessAVCmd* 则处理编码后的视频命令。

376

通过下列 3 个主要步骤,可将系统转换为物理实现[GVN94]:

- 1) **功能描述:**用语言来定义和描述整个系统所需的功能。功能描述包含计算和可能的时延关系,但不包含任何涉及物理实现的细节。
- 2) **系统设计:**将标准系统组件(如,处理器、存储器等)和定制组件(如,门阵列、FPGA、ASIC等)分配给设计。系统的功能划分到这些组件上,结果将产生每个组件的功能描述。
- 3) **组件实现:**每个组件的功能将根据其类型,通过硬件或软件实现。

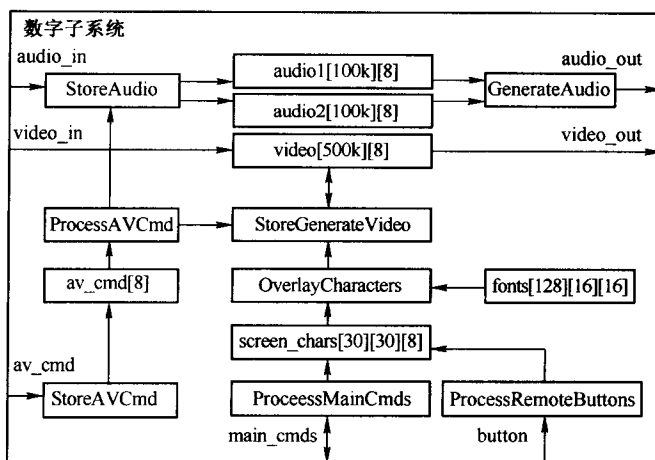


图 9-2 ITVP 中的主要行为和数据流

在完成前面两个步骤后,系统组件已得到定义,此时系统表示为设计者熟悉的“框图”形式。上述例子的框图如图 9-3 所示。系统由 6 个组件来实现:3 个存储器、2 个专用集成电路 (ASIC) 和 1 个处理器。图 9-2 中的每个行为和变量都只被分配到其中 1 个组件上。组件 Memory1 存储数组 *audio1* 和 *audio2*, Memory2 存储 *video* 数组,而 Memory3 则存储 *fonts* 数组和 *screen_chars* 数组。ASIC1 实现行为 *StoreAudio* 和 *GenerateAudio*, ASIC2 实现行为 *StoreGenerateVideo*、*StoreAVCmd* 以及变量 *av_cmd*。行为 *ProcessAVCmd*、*ProcessMainCmds*、*ProcessRemoteButtons* 及 *OverlayCharacters* 都在处理器上实现。

377

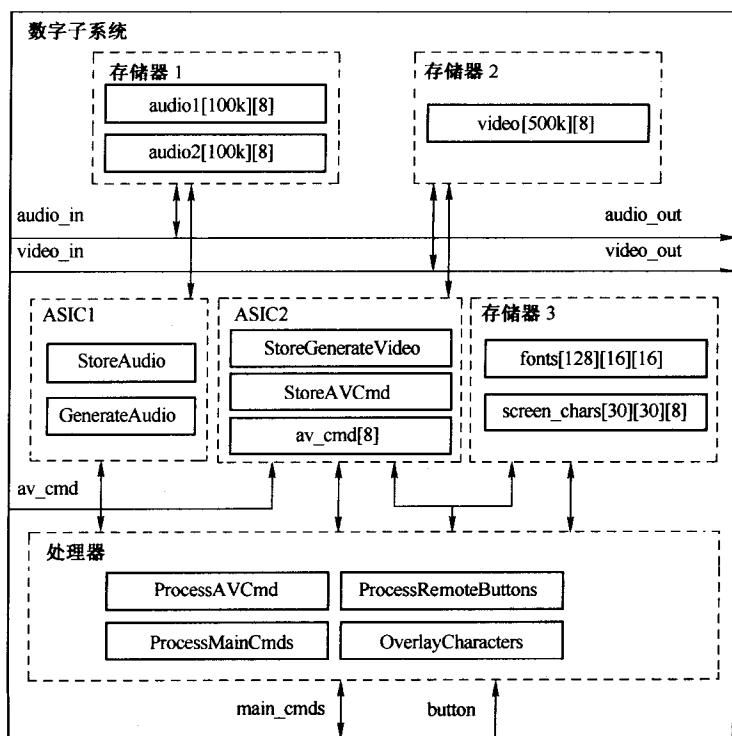


图 9-3 一种 ITVP 的设计方案

378

让我们考虑 ITVP 设计例子中所做的决策。由于音频和视频必须同时输出,因此用两个存储器来分别存放音频数组和视频数组。如果用同一个存储器来存储,则需要对存储器进行多路选择访问,这就可能违反最小音频/视频输出率的约束。行为 *StoreAudio*、*GenerateAudio* 和 *StoreGenerateVideo* 都在 ASIC 上实现,因为用软件实现可能无法满足输入输出速率的约束。由于一个 20 000 门的 ASIC 无法同时容纳音频行为和视频行为,因此它们实现于不同的 ASIC 上。行为 *StoreAVCmd* 和变量 *av_cmd* 也在 ASIC 上实现,因为音频/视频命令必须即时获得,这在处理器上很难做到。数组 *fonts* 和数组 *screen_chars* 不需要并发访问,因此将它们映射到同一个存储器上,同时也不会降低性能。映射到处理器上的行为的性能约束不高,因此可以用软件顺序执行,即使规定这些行为并发进行。

图 9-3 中的框图只给出了 ITVP 的很多实现方案中的一种。例如,两个 ASIC 组件可用一个更大的门阵列代替。或者,我们可采用不同代价和性能特性的 ASIC 工艺。我们甚至采用微控制器来代替处理器以降低成本。将行为和变量分配到给定组件集上有很多可行的方案。总的来说,系统设计就是针对不同的组件集、体系结构和工艺,枚举和探索可能的方案。

本章后续内容将就“什么是好的系统设计方法学?”和“这种方法学需要什么工具的支持?”这两个问题来展开讨论。

9.3.1 当前的惯例

我们现在来讨论当前在进行以上三个步骤过程中的惯例。图 9-4 给出了总结性的图示。

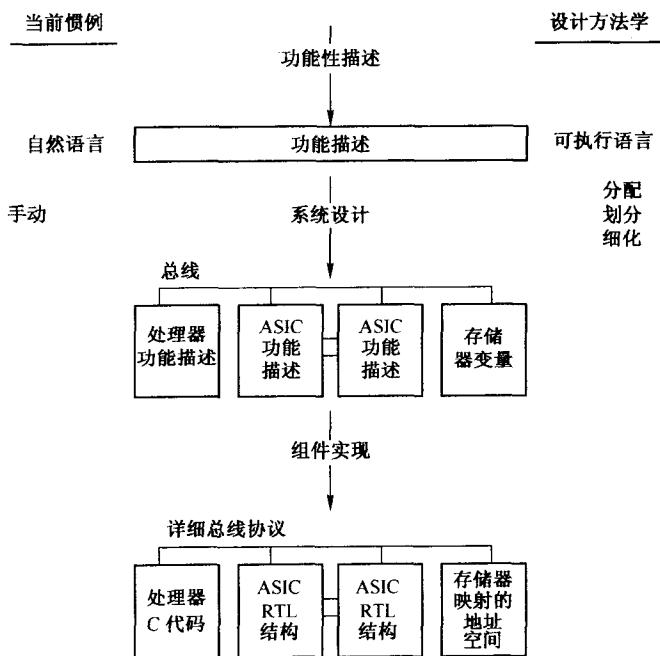


图 9-4 系统设计:当前惯例和所提出的设计方法学

功能描述:系统功能使用非正式自然语言(如英语)进行描述。有时,该描述还附带有数据流图、流程图、时序图表。从第 4 章中可以看到,很难建立一种既精确又易读的自然语言描述。这样,多数的自然语言描述为了保持可读性,给出的是非精确的描述。

系统设计:系统设计以手动方式进行,通常采用特定的方式,即设计决策是建立在先前经验或手工计算的质量评估的基础上的。手动设计方法学有如下几个缺点。首先,如第 6 章所示,人工考虑所有可能的设计方案耗时太长。其次,在评估中需要考虑很多细节,因此很难获得精确的评估结果,正如在第 7 章中所讨论过的。再次,对系统设计决策做出书面文档也是冗长耗时的工作,因而在多数情况下都被设计者所忽略。

组件实现:获得系统框图后,每个组件的功能都以自然语言描述。该描述用于设计每个组件的寄存器传输级或逻辑级结构。在某些情况下,用可执行语言来描述组件的功能,这样就可以进行功能验证。在很少数情况下,才会使用高层次综合工具从组件的功能描述来设计寄存器传输级结构。

总之,当前设计惯例很少注重功能描述和系统设计,而过分强调组件的实现。在系统级上,这种设计工作的缺乏可以归因于缺少得到检验的设计方法学和支持设计任务的工具。

9.3.2 系统级方法学

接下来对本书中描述的方法学进行讨论。同样如图 9-4 所示。

功能描述:系统功能使用可执行语言而不是自然语言进行描述。在系统设计开始之前,可以通过执行验证和模拟来确保正确的功能。通过这两步可获得对功能的极为精确的描述。

系统设计:系统设计包含 3 个明确定义的任务,分别在 3 类的功能对象上执行,如图 9-5 所示。在任何功能描述中都包含的 3 类功能对象是变量、行为和通道。变量用来存储数据,行为对数据进行转换,而通道用于在行为之间传输数据。在每类对象上,都执行 3 个任务:分配、划分和细化。

		系统设计任务		
		分配	划分	细化
功能对象	变量	存储器	变量到存储器	地址分配
	行为	处理器	行为到处理器	接口
	通道	总线	通道到总线	仲裁/协议

图 9-5 系统设计任务

分配为给定的功能描述定义系统组件。第 1 类系统组件包含存储器、ROM、寄存器文件以及寄存器。它们用于存储标量和数组变量。第 2 类组件包含处理单元,如标准处理器、微控制器及 ASIC 芯片。这些标准和定制的处理单元可用来实现行为。第 3 类包括物理总线,用来实现通道。

划分将功能对象分配到各个组件上。变量、行为和通道分别分配到存储器、标准/定制处理器和总线上,正如第 6 章中所讨论过的。

细化对原始描述进行修改,来反映给定的分配和划分的影响。划分到存储器上的变量需要存储器地址译码。不同组件上的行为需要进行修改,来保证它们之间的正确通信。分配到总线上的通道需要进行接口综合来确定通信协议,并且需要仲裁综合来解决对总线的并发请求问题。细化是第 8 章的主题。

上述 3 个任务将不含有任何实现细节的功能描述转换为一个新的描述,新描述包含一些结构信息,这些结构信息描述了系统级的体系结构。

382

应用上述 3 个任务的先后顺序并不固定。一种可能获得较好结果的任务顺序如图 9-6 所示。功能被描述后,大量变量被映射到存储器,并且接近度好的变量(定义见第 6 章中对变量接近度的定义)将共享同一个存储器。通道用类似的方式映射到总线。接着分配处理器或 ASIC 组件,将行为(以及没有分配到存储器上的变量)划分到这些组件上,并使得软件和硬件的总成本最小。此后,还可能对变量或通道进行再划分,以进一步降低成本。最后需要进行接口综合和仲裁综合,以获得系统中每个组件的完整的功能描述。

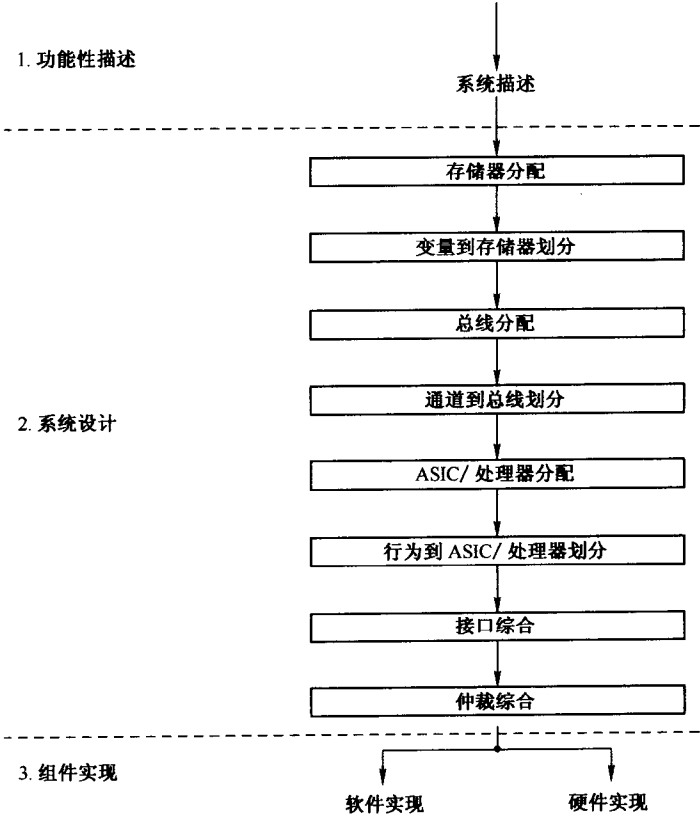


图 9-6 一种系统设计任务的安排顺序

383

组件实现:通过将组件的功能描述编译为机器代码可以实现软件组件,而 ASIC 组件通过手工设计或采用 CAD 工具进行综合。由于组件是形式化定义的,故采用综合更自然。

9.4 通用综合系统

这一节里,我们提出一个假想的通用综合系统,用于嵌入式系统设计中从概念到制造的变换。该系统的通用性体现在它集结了前面章节所讨论到主要概念。该系统是假想的则是因为现有的研究或商业综合系统都不能全部满足下面的准则:

- (a) **完整性:**系统需要提供设计过程中所有级别上的综合工具,包括从描述到制作文档。而且这些工具应能适应多种实现风格。
- (b) **可扩展性:**系统必须足够灵活,允许增加新的算法和工具,以及对新的实现风格的支持。

- (c) **可控制性**:对特定的描述,设计者应可以控制所要应用的工具类型以及执行顺序。设计者还应该能通过选择组件、拓扑、体系结构和工艺来控制对设计空间的探索。为了帮助设计者做出选择,系统还应提供多种设计质量度量和权衡的线索。
- (d) **交互性**:设计者应能和综合工具进行交互,主要通过部分地规定设计结构,以及在综合后对设计进行修改来实现。
- (e) **可升级性**:综合系统应该允许方法学从“捕获-模拟”到“描述-综合”的进化升级,并且允许在每个抽象级别上混合使用两种策略。

该通用综合系统的总体框图以[GDWL91]中描述的一个类似系统为基础,如图 9-7 所示。该系统是完整的,因为它支持在系统、芯片、逻辑及物理等级别上的综合,包括软硬件协同设计。该系统含有一个综合工具,用来将可执行描述划分为一组描述,其中每个都用定制、半定制或者标准组件来实现系统功能。软件综合工具将处理器的描述转换为标准代码,该代码可以编译为处理器的指令集。ASIC 综合工具把芯片的描述综合为一组寄存器传输级组件。一个时序和逻辑综合工具将 FSM 和逻辑描述转换为门级网表。下面对每个综合工具进行详细介绍。

384

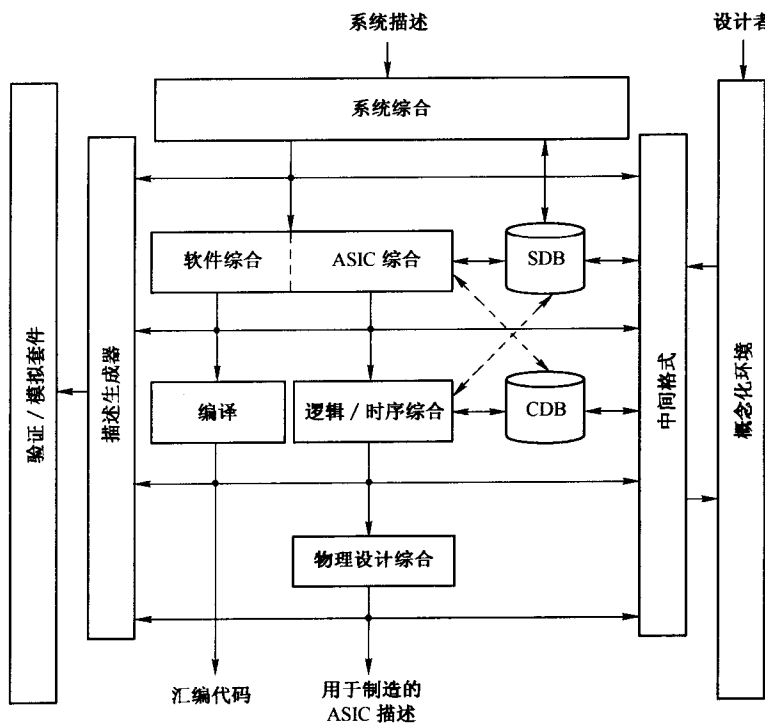


图 9-7 一种通用综合系统

该通用系统具有一个概念化环境来支持交互性和可控制性。扩展性由两个数据库来支持。组件数据库(Component database, CDB)支持增加新组件以进行 ASIC 综合,而系统数据库(System database, SDB)则支持设计工程和管理。

385

系统同时支持“捕获-模拟”和“描述-综合”方法。通过在概念化环境中捕获原理图并使用模拟套件中的某个模拟器进行模拟,可以实现对第 1 个方法的支持;对第 2 个方法的支持则是通过对以一种或多种捕获语言表示的设计描述进行捕获,并采用适当的综合工具进行综合

得到的。概念化环境提供了多种捕获语言。

概念化模型在不同的设计角度和设计阶段上采用不同的中间格式。捕获的设计描述被转换成某种综合和模拟所接受的一种标准中间格式。可以对所有抽象级别和设计风格定义一种通用语言,来代替这许多种中间格式。但是,这样一种语言可能过于笨重和低效。而且,需要很长的时间才能形成标准,可能需要更长的时间才能让设计者学会如何使用。另一方面,我们可以采用 VHDL 之类的标准模拟语言。然而,为支持模拟所需的结构是这种语言的负担,而且该语言很难用于捕获设计的其他方面。鉴于上述原因,采用特定的中间格式是便利的,每种特定的中间格式都接近于设计者对问题的观察视角,捕获特定的设计信息,只要我们提供一个模拟代码生成器来将中间格式转换为可模拟的描述。具有该用途的描述生成器如图 9-7 所示。

代码生成器的概念符合“捕获-模拟”方法。例如,可以将示意图看做中间格式,通过网表生成器转换为可模拟描述。与此类似,逻辑、时序、寄存器传输及系统级上的中间格式可转换为 VHDL 或其他模拟语言。这种方法使被捕获的描述更接近于设计者的思维方式,并且对于具有描述生成器的任何标准模拟器来说是可模拟的。此外,这种中间格式可用于交互式综合以支持对描述和设计的手动分配、划分、建立接口、仲裁、变换、验证,以及对设计质量和设计变化对质量的影响进行评估。

386 接下来将该通用综合系统中的每个部分进行简要介绍。

9.4.1 系统综合

系统综合接受整个系统的可执行描述,将其划分为一组可执行描述,其中每部分都用定制、半定制或标准系统组件来实现。每个组件描述都满足该组件的所有约束。例如,如果某个组件是一个 50 000 门的门阵列,含有 100 个引脚,则组件描述的实现必须少于 50 000 门,并且用于数据通信和电源的引脚也必须少于 100。同时,如果组件是标准存储器,则组件的描述必须包含存储器访问数据的协议。与此类似,对标准处理器,组件的描述在编译为机器码后,必须在给定的约束时间内执行完毕。

一种系统综合环境如图 9-8 所示,包含一个带相应系统表示(SR)的编译器和一组系统设计工具。分配器(Allocator)选择设计中所需系统组件的类型和数量。划分器(Partitioner)将 SR 的每部分分配到组件上。如第 6 章所述,分配器和划分器都会在 SR 中加入自己生成的新信息。评估器(Estimator)快速地向分配器和划分器提供各种质量度量的评估值,如第 7 章所述。接口综合工具在系统组件间加入通信总线的协议描述。仲裁综合工具对并发总线的访问增加新的描述。最后两个工具在从原始系统描述到组件描述的转换过程中具有本质上的重要性,这点在第 8 章里进行了介绍。

变换器(Transformer)工具是图中唯一一个前文没有提及的内容。它执行一组 SR 变换,以便在最终设计

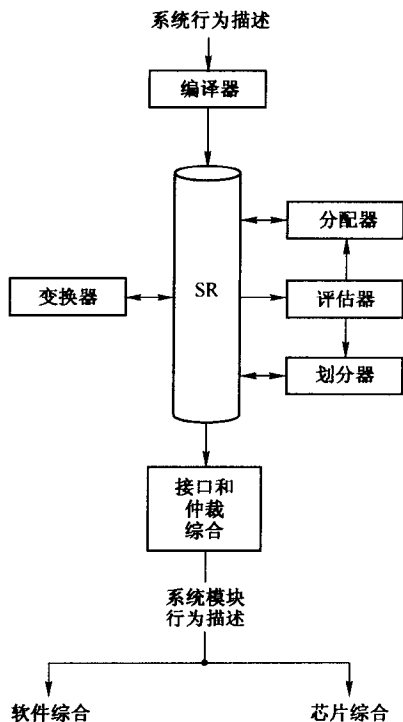


图 9-8 一种通用的系统综合环境

里改善某些质量度量。SR 需要进行变换的原因是它得自于具有可读性却不易于划分的描述。例如,过程 P 可能用于封装一个复杂的计算,该计算在描述中的多处使用。如果用单个 FSM-D 来实现 P ,在其他多个 FSM-D 调用 P 的情况下,将导致大量的引脚数量或者很差的性能。在这种情况下,对 P 进行复制并将其加入到每个调用它的 FSM-D 中也许是一种较好的方案。另一种更好的方案是内嵌 P ,这将会导致冗余的硬件,但能够提高性能并减小互联开销。其他可能的变换包括:(a)展开循环,以暴露更多的并发性。(b)将数组变量分为一组标量变量,或者将行为分为一组更细粒度的行为,以此来提供更多的划分选择。(c)将并发行为顺序化以减少所需的 FSM-D 数量。(d)将顺序行为并行化以提高性能。由于一个特定的变换可能促成或限制一个或多个其他变换,因此这些变换间具有很强的相互联系。

尽管系统级变换领域的研究显示出很有前景的结果[TAS93, HT93, WT89, IOJ94],但变换在系统设计中的角色仍不是特别明确。描述变换在未来将成为系统设计中的第四个主要任务。

9.4.2 ASIC 综合

系统综合定义了一组具有固定接口的系统组件,以及每个组件的可执行描述。芯片综合通常也叫做高层次综合(high-level synthesis, HLS),它将组件的描述转换为寄存器传输级组件(register-transfer component, RTC)的结构,这里的 RTC 包括寄存器、多路选择器和 ALU 等。这种结构通常包含两个部分,其一是控制器,用来实现有限状态机;其二是数据通路,用来执行算术操作。我们称这种结构为带数据通路的有限状态机,或者 FSM-D [GDWL91]。控制器控制数据通路中的寄存器传输,产生同外界通信的信号。根据 [BM89, BCD⁺ 88, CR89, TLW⁺ 90, CST91, Gaj91, KD91, LND⁺ 91, NON91] 中的思想得到的通用芯片综合环境如图 9-9 所示。

综合环境包含一个带有相应表示方案的编译器,一组 HLS 工具,一个 RTC 数据库,工艺映射器及一个优化器。输入的可执行描述首先被编译为一种称为控制/数据流图(control/dataflow graph, CDFG)的设计表示,它表示出基本操作(如加法和比较)之间的控制和数据依赖关系。例如,图 9-10 显示出一个简单的行为和相应的 CDFG。需要注意的是,加法操作“ $a + b$ ”和比较操作“ $a = b$ ”即使在描述中以顺序方式书写,也可以并行执行。而减法操作“ $x - y$ ”必须在“ $a + b$ ”之后进行,因为 x 依赖于“ $a + b$ ”的计算结果。

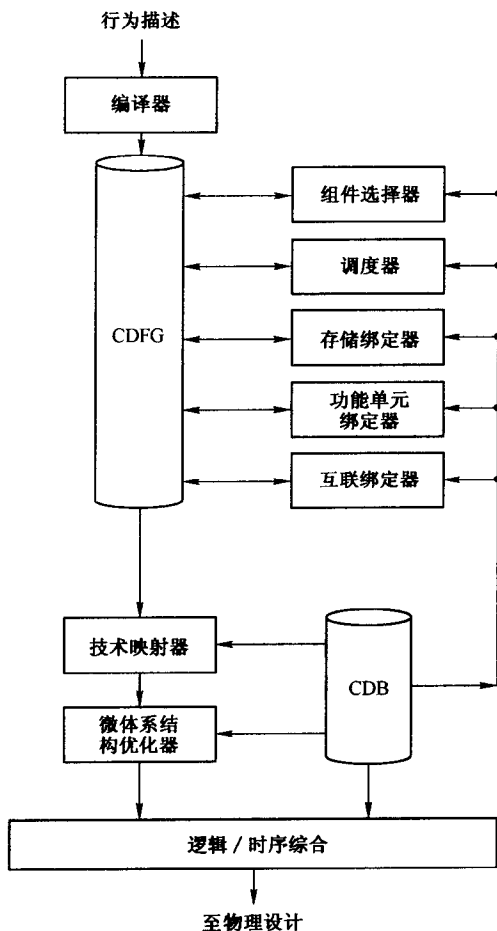


图 9-9 一种通用的芯片综合环境

389
390

一些高层次综合工具在建立寄存器传输级结构的时候对描述进行标注。组件选择器从 RTC 数据库中选择设计中使用的存储、功能及总线等单元。调度器将操作映射到控制步,其中每个控制步通常代表一个时钟周期或时钟相位。由于数据依赖关系或执行特定操作的单元数量有限,所有操作通常不可能立即执行,因此需要进行调度。在该例子中,加法和比较操作如果使用不同的 ALU,就可以在一个控制步里执行。条件“ $a = b$ ”可在第 2 个控制步检查,接下来两个控制步里进行减法和小于操作。注意由于减法和小于操作存在数据依赖关系,因此不能在一个控制步里执行。调度是 ASIC 综合研究中的一个核心课题之一,已有许多不同的算法[GDWL91]。存储、功能及互联等单元绑定器将标量和数组变量、操作以及互联分别映射到寄存器和存储器、功能单元以及总线上。在上述例子中,可以选择两个 ALU,将“+”和“-”操作、“=”和“<”操作分别分配到其上。虽然多数研究都假设使用一组基本的寄存器传输级组件,近年来的研究则关注于使用更加实际的组件库。

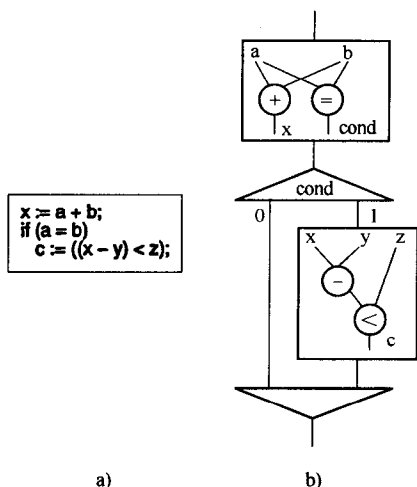


图 9-10 CDFG 例子

391

显然,对于一个特定的描述,选择、调度或绑定的方案不是唯一的。例如,上述例子可以只使用一个 ALU,这样“+”和“=”操作总共需要两个控制步来执行,而不是一个。也可以将“-”操作和条件判断“ $a = b$ ”放在同一个控制步里执行,并且在比较操作判断为假时抛弃“-”操作的结果。这些只是不同选择、绑定及调度的例子中的两个。不同的选择、绑定及调度决策所得到的设计主要在面积和性能上有差别。有时,面积约束优先级高于性能。在这些情况下,设计目标即为选择满足面积约束的组件,并获得最好性能。另外一些时候,性能约束优先级较高。在这些情况下,设计目标则为在满足性能约束的条件下,选择面积最小的一组组件。

需要明确的是选择、绑定及调度这些任务之间有很强的关联。如果先进行选择 and 绑定,就限制了可能的调度。例如,两个操作绑定到同一个组件上,就不能调度到同一个控制步内。另一方面,如果先对操作进行调度,则会限制选择和绑定。例如,将两个操作调度到同一个控制步内,则它们就无法共享一个组件。在[CT90, BM89, Geb92a]中,研究者尝试将选择、绑定和调度合并到一个算法里,以此来处理这三个任务间的相关性。这三个任务中的每一个都是 NP 完全问题。

组件数据库(component database, CDB)储存寄存器传输级组件,可供综合过程中使用,并可对它们的特性进行查询。工艺映射器将通用组件从综合后的设计描述映射到 CDB 中的组件实例。由于映射到实际组件后,评估得到的关键路径及其上的延迟可能有所改变,因此使用微体系结构优化器来减小关键路径上的时延,做法是重新分配组件,在关键路径上插入更快的组件,在非关键路径上插入较慢的组件。

392

CDB 中的某些组件,如存储器和多路选择器,通过标准宏单元来实现。其他组件,如 FSM,则从行为描述综合而得到。在物理设计过程中,它们共同组成一个 ASIC。

9.4.3 逻辑综合和时序综合

芯片综合产生一组控制器和数据通路。每个控制器用有限状态机(FSM)建模。控制器综合工具必须将 FSM 转换为硬件结构,包括一个状态寄存器和用于产生次态及控制器输出的组合电路。生成这种结构的任务包括状态最小化、状态编码、逻辑最小化及工艺映射。基于 [DSVA87,BRSVW87,DMNSV88]中思想的通用逻辑综合系统如图 9-11 所示。

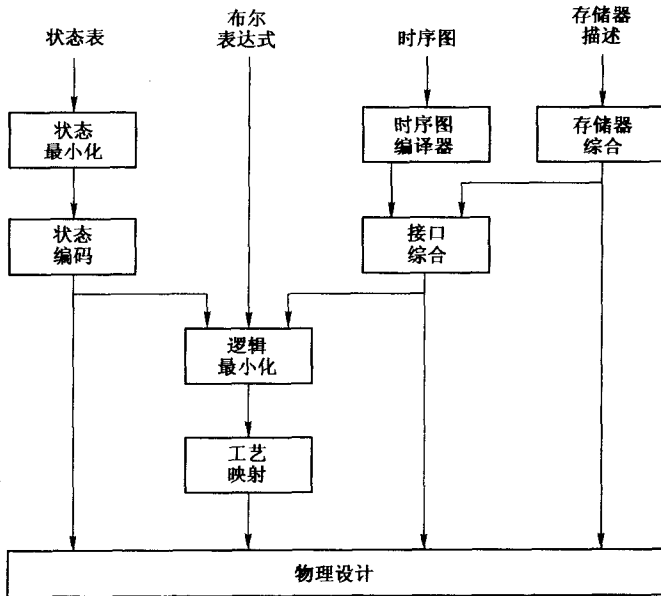


图 9-11 一种通用逻辑综合工具

状态最小化是用一个状态来代替所有与其等价的状态,以减少 FSM 中的状态数。所谓两个状态等价,是指对于任何输入序列,无论我们从哪个状态开始,输出序列都相同。状态最小化的重要性体现在状态的数量决定了状态寄存器和控制逻辑的大小。

状态编码对符号状态分配二进制代码,其目标是最小化编码控制器的组合逻辑。逻辑最小化用于在编码后减小组合逻辑的面积或时延。

两级组合逻辑通常用 PLA 来实现,而多级逻辑可用标准逻辑库实现。工艺映射接受逻辑最小化器产生的与工艺无关的逻辑网络,将其转换为由特定库中的标准门组成的网表。

由于我们假设每个设计都可以用一组相互通信的进程来描述,并采用一组相互通信的 FSMD 来实现,因此设计通信逻辑是综合的一个必需的部分。接口电路可以实现成为每个 FSMD 的组成部分,也可以用独立的 FSMD 来实现。如果通信协议是 FSMD 描述的一部分,那么由于输出信号只在状态的边界处发生变化,协议信号的响应就需要整数个状态或时钟周期。例如,对一个请求-确认协议,可以用一个等待状态和一个确认状态来实现。请求信号的到来使 FSMD 由等待状态进入到确认状态,并导致确认信号变为有效。当请求信号被撤销后,FSMD 将离开确认状态。另一方面,当两个具有不同协议的标准 FSMD 必须建立接口时,就必须综合出另一个 FSMD 来将一个协议转换成另一个协议[Bor91,NT86]。

由于缺乏接口描述语言和针对同步/异步混合时序逻辑的综合方法,接口综合是一个难点。通信协议通常用带时间约束的时序图来描述,如商业存储器和微处理器手册中给出的存

存储器读写周期描述。接口综合包括将时序图映射到最少数量的锁存器和触发器,减少设置和再设置需要的逻辑。根据标准组件支持的协议,接口逻辑可以是同步的或者是异步的。

394

存储器综合根据给定的存储器需求生成存储器的结构,这里的需求包括字数、每个字的位数、端口数、端口类型(读、写或读/写)、访问速率、访问协议及时延约束等。对某些需求,存储器综合可能会很复杂。例如,只用单端口存储器芯片来设计一个四端口的存储器,如果数据能分为四组,且每组只通过一个端口来访问,那么综合就会很简单。然而,如果同样的数据需要通过两个不同的端口来访问,或两个端口访问同一个单端口存储器芯片的数据,那么就on需要引入解决冲突的逻辑,从而增加存储器的访问时间。

9.4.4 物理设计

执行上述综合步骤之后,设计的硬件部分就被描述成为逻辑级和寄存器传输级组件的网表。大量的工艺可用于这种网表的物理实现,每种工艺都有自己的设计方法。比如,网表可以用定制版图来实现。这种实现的设计问题包括晶体管布局 and 定向、晶体管大小改变以及布线。诸如8位ALU等规则结构可以采用模块生成器来实现,并通过利用规则性来获得非常紧凑的版图。如果采用标准单元来代替定制版图,则单元的布局和布线是关键问题。同样,也可以使用模块生成器。

一种常用的方法是采用现场可编程门阵列(FPGA),其中含有数百个模块阵列和接线开关盒,以及组合/时序逻辑[Xil89]。逻辑块有 n 个输入和 m 个输出,其中 n 和 m 是2~10之间的整数。这些模块和开关盒可由设计者在现场编程,克服了标准单元和定制方法设计周期长的缺点。设计问题包括将逻辑表达式分解为一个 n 输入的表达式,及将该 n 输入表达式分配到逻辑块并使得门阵列具有可布线性(routable)。

9.4.5 软件综合

395

可执行描述与C等传统程序设计语言相比,通常具有一些独有的特点。典型的编译器通常不能处理这些特点。软件综合的任务是将复杂的可执行描述转换为传统软件程序,使其可以用传统的编译器来编译。

可执行描述的特点之一是并发任务的定义。如果两个并发任务映射到一个处理器上,则必须经过调度来顺序执行。这种调度通常称为多道程序设计[HB85]。在调度中,重要的是确保每个任务都有机会执行,即没有任务“饿死”。另一个问题是最小化处理器等待外部事件所需的时间,即最小化“忙-等”。第三个问题是保证满足每个任务的时间约束。例如,数据以特定的速率到达,必须被指定的任务获得并处理,或者为保证系统性能,某任务必须以某种速率输出数据。这种任务必须确保最小的执行速率。

现有若干种技术能处理这种调度[HB85, GD93, AS83]。其中一种使用全局任务调度器,通过类似子例程调用的方法来激活每个任务(或其中一部分)。这种技术在任务之间切换的时候,需要增加开销来保持每个任务的状态。另一种技术在每个任务里保持局部数据来记录状态,并且在任务需要等待事件或中断的时候将任务改为不接受处理器控制,以此来减小上述开销。不同技术的选择通常涉及性能和程序大小之间的折中。

有的时候,我们可能希望将一组并发任务映射到多处理器上。在这种情况下,就必须涉及与多机处理相关的问题[HB85]。

对于这种必须转换为可编译格式的描述,另一个普遍特点是使用时间相关的语句,如

VHDL 里的 wait 语句或 HardwareC 里的并发语句。这种语句通常被转换为一组等价的传统程序语句,并可能增加一些额外变量,这一点在第 5 章里进行了讨论。

396

9.4.6 系统数据库

传统的综合系统开始于一些松散的和相互独立的设计工具。这种系统存在数据表示及输入/输出格式的不匹配问题。第二代综合系统为紧密或松散集成的系统。紧密集成的系统采用公共数据表示,所有工具都使用同样的过程来访问。这种系统效率很高,但缺乏灵活性,因为格式的一点变化就会影响到所有的工具。松散集成的系统将设计数据与工具分离。数据存储在数据库中,每个工具只访问它需要的信息,并使用自己的表示来处理检索的信息[CT89, LGP⁺91, LND⁺91, RG91]。

在设计探索中,数据库还支持同一个设计的多种不同版本。这种支持减轻了设计者管理设计修改和配置的负担。在系统数据库的支持下,系统的每个版本及与以前版本不同之处都会被记录下来。在任何时候都可以放弃所做的工作,回溯到任何一个版本。版本间的不同之处可以输出作为设计决策的文档。可选择设计的不同质量度量可以输出,以帮助在设计间做出折中决策。设计的不同部分可以同时被多个设计者修改,而由数据库来管理数据的一致性。

9.5 系统设计的概念化环境

从系统描述开始的完全自动设计过程是一个虽然目前还不现实,但却重要的目标。即使已拥有所有的系统和芯片级工具,设计者也需要熟悉并有信心去使用它们。综合工具在不同风格的设计中获得始终如一的质量也需要花费时间。因此,有必要提供一个环境,使得设计者可以控制设计决策,并采用手工设计来代替自动综合和优化。即使当综合算法已很完美,也要允许设计者做高层次决策,如选择系统组件、实现风格,以及约束。这样,概念化模型必须允许在设计过程中每一阶段对设计决策和设计策略进行控制,这就进一步要求环境在每一个抽象级别上提供可用度量的快速反馈,并允许设计者选择组件、拓扑、实现风格、优化目标、成本函数、设计约束、工艺映射和划分[BE89, CPTR89, YH90, HG91]。概念化环境与系统和组件数据库协同工作。这些数据库存储了以往长时间内各个版本的设计数据,其中概念化环境只访问其中很小一部分数据,用于短时间内的修改和细化。概念化环境可看做设计者在设计细化过程中用到的一个短期存储器(便笺式存储器)。概念化模型(见图 9-12)包含以下几个部分:设计管理器、质量评估器、综合算法、设计一致性检验器、显示器和编辑器。

397

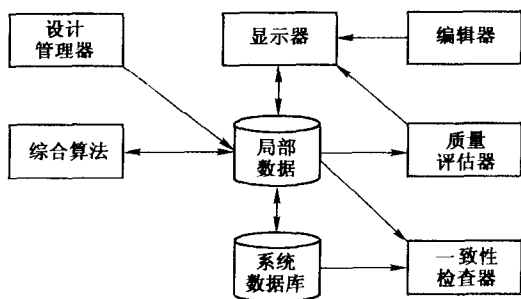


图 9-12 系统设计的概念化环境

设计管理器用于维护局部设计数据,例如当前组件的分配和对象的划分。质量评估器提供设计度量的快速反馈,如第 7 章所述。综合算法对设计做出实际的改动。他们可以应用于整个设计或其中某些部分。设计者应能将综合分为数个较小步骤,对每步分别应用算法。在手工设计过程中,综合工具可以看作为设计者提供一些提示。例如,在划分过程中,设计者可以每次把一个行为分配到系统组件上,划分算法能给出每个行为的最佳映射的建议,设计者可

398

以接受或拒绝。设计一致性检查器用于确定手工设计的修改没有改变系统的功能。例如,当设计者为了将两个并发表行映射到一个串行处理器上,而将其改为顺序执行时,一致性检验器必须保证这种顺序化的行为和原来的并行行为产生相同的结果。

显示器和编辑器使得设计数据能被显示和编辑。这二者都包含通常所指的用户界面(user interface, UI)。UI 是概念化环境的一个重要部分,因为设计者只有在易懂和有用的数据显示和编辑环境下,才能有效地使用该工具。没有一个好的 UI,即使是最好的综合算法也发挥不了作用,因为设计者无法决定何时应用这些算法。对于前面描述的系统设计任务,一个好的 UI 必须以易于快速理解的方式对总体设计状态进行概括总结,可作为设计文档,并突出强调设计中需要特别注意的部分(如违反约束)。总体设计状态包含可执行描述、当前已指派的组件及其连接性、当前功能对象到这些组件的映射情况以及对所选择的质量度量的评估和约束。

给定总体设计状态,设计者可以对很多途径进行探索。设计者可以对有关评估方式的细节进行检查。如果不同意评估器所做的假设,则可以不采用该评估。设计者可以修改约束,也可以修改设计状态的显示,以关注于某一部分设计或某个度量。设计者还可以手工执行诸如分配、划分等设计任务,以及要求诸如单个对象的最好可能放置、调度、绑定等的提示信息。

系统设计工具的用户界面的一个例子如图 9-13 所示。其显示的信息对应于图 9-3 中框图的划分。映射这一列显示出功能单元到所分配的系统组件的映射。例如,组件 ASIC1 包含行为 StoreAudio 和 GenerateAudio。组件类型这一列指明了每个系统组件的类型,即系统组件代表的库组件。从组件库中这个类型可以获得该组件所有的信息。例如,系统组件 Processor 绑定到类型为 Y900 的处理器上。余下的几列表示选择的质量度量。一列中的每一项给出了相应质量度量的评估值和需要值。成本这一列给出了每个被分配组件的成本。执行时间这一列显示出执行时间的评估值。面积和引脚列则提供了每个 ASIC 组件的面积和引脚数。指令数列给出了每个处理器机器指令数的评估值。违反约束的情况用星号标明。为了简明起见,图中略去了总线和通道。其他质量度量(见第 7 章)也可以增加到显示中。

映射	组件类型	成本	执行时间	面积	引脚	指令数
系统		105/100*				
ASIC1	X100	30		16k/20k	46/60	
StoreAudio			100/110			
GenerateAudio			100/110			
ASIC2	X100	30		18k/20k	48/60	
StoreGenerateVideo			100/110			
StoreAVCmd			100/110			
Memory1	V1000	10				
audio_array1						
audio_array2						
Memory2	V1000	10				
video_array						
Processor1	Y900	25				6k/5k*
ProcessRemoteButtons						
ProcessMiscCmds						
Cost: 5.43						
视图选择 划分 / 分配 细化						

图 9-13 系统综合工具的用户接口

为了加强设计者的理解,需要提供设计的其他视图方案。这些视图方案包括只选择一些质量度量来显示,显示代价函数表达式的细节,或者观察有关如何获得特定质量度量值的详细信息等。这些视图可以保存到文件中以用作设计文档。例如,图 9-14 中给出的视图以图形方式显示了图 9-13 中受约束质量度量的评估和约束。设计者可利用这些信息来决定下一步的设计着重点。例如,图中违反约束最主要的是处理器程序指令数。注意到两个 ASIC 的面积都接近组件的容量,设计者可以选择更大的程序存储器,而不是将行为从处理器移动到 ASIC 上。

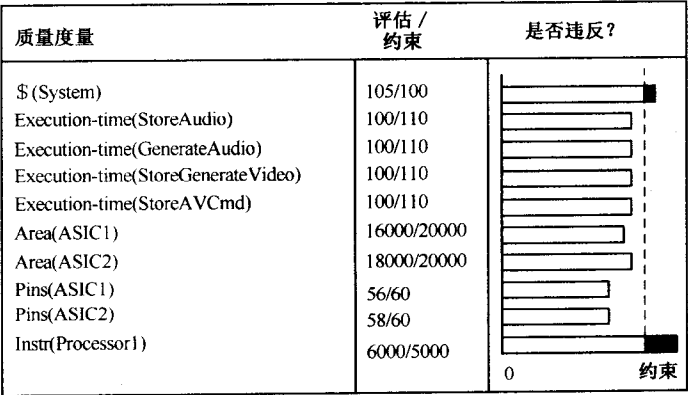


图 9-14 可选视图,只显示评估和约束的关系

图 9-13 中还有几个按钮,分别用来激活划分、分配和细化工具。划分和分配工具允许浏览和选择库中组件,对其进行分配,应用自动划分和手工移动等方式将功能对象从一个组件移到另一个组件。它还提供了一些提示信息,如第 6 章讨论的各种接近性准则的基础上,给出每个功能对象与选定对象间的接近性值的一个列表。细化工具允许选择通信协议和仲裁方案。

401

概念化环境允许使用多种不同的方法学。设计者应反复多次处理同一个任务,直到满足需求为止。很显然,设计者希望做出大多数的高层次决策,而只手工对设计中的小部分进行综合。设计的剩余部分都由自动综合工具生成,这样就不会对设计者所做的高层次决策有太大的影响。

总之,对于所有设计者在高抽象层次上所完成的任务,概念化环境允许设计者手工执行同样的任务。然而,概念化环境还允许设计者使用自动综合工具来完成部分任务。随着算法的改进,更多的设计工作将可以自动实现。

9.6 结论和发展方向

这一章里,我们介绍了一个三步的设计方法学,包括系统描述捕获、系统设计和组件实现。前两步是 CAD 研究领域,也是本书的研究焦点。我们定义了系统设计步骤中的主要任务,讨论了在系统、芯片、逻辑和物理等级别上的设计和工具问题。我们还简要介绍了系统设计的支持环境,包括系统数据库和用于交互和自动综合的概念化环境。未来的研究领域包括如下几个方面:

- a) 针对新硬件工艺、实现方式和体系结构研究新的评估方法。新的评估算法必须能预测软件和硬件优化技术的结果。
- b) 引入形式验证技术,来提高设计的正确性,减少目前在模拟各阶段设计过程中所需要

的大量时间。

402

- c) 在设计过程中尽早引入可测试性设计,并在每个设计阶段进行可测试性度量的计算。
- d) 必须开发相应的框架和数据库来管理系统级的工具和数据。
- e) 必须研究系统级的转换技术并将其加入到现有的设计任务中。
- f) 设计实现后的质量度量必须反馈给设计过程的高层次,以支持设计迭代反复。
- g) 用以支持手工控制综合过程的交互式综合技术也需要研究。
- h) 通过定义设计质量度量,并了解设计风格、结构拓扑关系以及设计方法学对设计质量的影响,会极大地有助于设计探索的自动化。这样,对不同级别上的设计质量进行比较分析将是非常有益的。

总之,不仅在算法研究方面,而且在系统和 ASIC 综合所需基本体系结构的开发方面,还需要大量的工作。我们希望本书能够作为这方面研究的一个起步。

9.7 练习

- 1. 说明采用自然语言进行系统功能描述的设计方法学的局限性。列举采用可执行系统描述语言的优缺点及其对方法学的影响。
- 2. 给出图 9-6 中系统设计任务的一种顺序,使所产生的系统的组件间连线数最少。
- 3. 给出一种从系统描述开始进行系统综合的系统设计过程,其中的组件只包括 1 个处理器芯片和 1 个存储器芯片。

403

- 4. 给定功能描述所需的总执行时间,设计一个算法,从功能描述中选择最小成本的组件集合。
- 5. 扩展上一题的算法,产生形状函数,在给定性能约束范围的条件下分配最小成本组件集合。
- 6. 列举将通道划分到总线的 3 个成本函数。
- 7. 设计一个算法,将并发行为调度到两个处理器上执行。
- * 8. 给出一种技术,在时延约束下顺序化并发行为。
- * 9. 给出图 9-8 中转换工具的需求。
- * 10. 列举系统设计过程中需要设计者交互的所有阶段。定义需要显示给设计者的度量和设计者要输入的设计参数。
- * 11. 对图 9-7 中的系统数据库,设计一种一致性检测方案。
- * 12. 除本章中提到的以外,再定义几个有用的系统级转换方案。
- * 13. 对图 9-7 中的组件数据库定义其信息模型。给出不同设计阶段数据库提供的信息类型和数据库响应的查询类型。
- * 14. 对比图 9-13 中的文本显示,描述并示范一个交互式划分的图形显示。当前的划分如何显示? 如何显示评估和约束? 设计者如何分配组件和移动对象?
- * * 15. 设计一种技术,在系统或 ASIC 综合中采用真实实现的度量,以此来支持综合和实现间的设计反复。
- * * 16. 设计一种方法,将详细时延信息结合进系统综合中。

404

附录 A 应答机的自然语言描述

本节收录了在第 4 章中出现的电话应答机控制器的自然语言功能描述。控制器的 Spec-Charts 形式的系统描述就是从这个描述中直接产生的。为了便于引用,我们将每一句描述编号。控制器的环境和接口已经在 4.2 节的图 4-1 中给出。

对外部按键和开关的响应

1. 当“power”开关处于“off”位置时,机器忽略对电话线和所有的按键响应。
2. 当为“on”时,显示模块显示当前的信息数,初始为 0。
3. 按下“play-messages”按键将播放所有的消息。
4. 常见的录音机按键都有:播放 play,快进 forward,快退 rewind 和停止 stop;按下任何一个都将使当前的信息计数归零。
5. 按下“record-announcement”按键将触发一声蜂鸣,几秒钟后还有一声蜂鸣。
6. 在两声蜂鸣之间通过麦克风录制的任何声音都将成为预录的通知。
7. 按下“play-announcement”按键就能听到通知。
8. 按下并保持“memo”按键将允许用户通过外部的麦克风来录制一段消息。
9. 按键松开时消息录制中止。
10. 因此“memo”按键对其他家庭成员留言是有用的。

405

对电话线的响应

A. 监测线路上的响铃

11. “on/off”按键用于将机器从“开”状态置为“关”状态,或者相反。(当机器处于“off”,控制器不能响应呼叫,然而当“power”为“off”时,控制器将不能对任何的输入产生响应。)
12. 当为“on”时,显示消息数目的模块下方的发光二极管被点亮。
13. 当为“on”时,机器一般在响铃四次后应答。
14. 然而,如果“tollsaver”为“on”且至少记录了一条信息,则机器在铃响两声后应答。
15. Tollsaver 允许机主通过电话来判断是否有信息记录下来。
16. 如果电话铃响三次,则说明没有信息记录,机主可以挂断,这样就避免了长途通话费用。
17. 有时机主在离开寓所之前可能会忘记打开机器。
18. 因此,即使是处于“off”,机器也将在响铃十五次之后应答。
19. 因此机主可以按下面的方式来远程打开机器。

B. 正常的应答活动

20. 一旦机器响应来电则会播放通知。
21. 当通知播放完毕,会自动响起一声蜂鸣,同时来电线路上的信息开始被记录下来直到呼叫者挂断或者超过最大的消息时间。
22. 机器再次挂起并监测响铃。

406

- 23. 如果在播放通知的过程中挂断,则机器立即挂起并且不进行消息的录制。
- 24. 如果在播放通知或者录制信息的过程中用拨号音 1,则机器立即进入远程操作模式。

C. 远程操作的应答活动

- 25. 远程操作模式的第一步就是检查用户的身份号码。
- 26. 接下来的四次按键音的数字将送入机器和内部存储的四位号码进行比较。
- 27. 如果不匹配,则机器挂断电话。
- 28. 如果确实匹配,机器进入基本命令 *basic-commands* 模式,在该模式下它可以被指示执行几个基本命令。
- 29. 拨号 2 将播放所有的消息。
- 30. 拨号 3 到 6 对应标准的录音机按键:播放 play,快进 forward,快退 rewind 和停止 stop。
- 31. 拨号 7 将触发蜂鸣,蜂鸣的数量代表录制的信息数。
- 32. 拨号 8 将指示机器离开基本命令模式进入复杂命令 *miscellaneous-commands* 模式。
- 33. 在该模式下,拨号 3 将清除“erases”所有录制的信息,即快退到录音开始,并且设置消息计数为 0。
- 34. 拨号 4 将播放通知。
- 35. 拨号 5 和按下“record-announcement”按键一样。
- 407 36. 拨号 6 将设定机器的开关状态在挂断之后从“off”变为“on”或者相反。
- 37. 拨号 2 将指示机器回到基本命令模式。
- 38. 在上述两种命令模式的任一种发生挂断,磁带都将重置到最后一条信息的结尾,即任何已经录制的信息都将保留。
- 39. 在基本命令模式时拨号 9 也将触发这种复位。
- 40. 在复位后或者在检测用户身份号码时呼叫方挂机,机器都将进入挂起状态。

复杂需求

- 41. 如果在机器应答之后“on/off”键被按下,任何当前的活动都将中止,同时机器将如前所述监测电话线。
- 42. 该功能用于屏蔽呼叫,机主可以监听消息并且随时拿起电话按下“on/off”键将应答机置为“off”状态,同时开始和呼叫者通话。
- 408 43. 在机器应答电话的时候应答机的按键具有优先权;因此,按下任何外置按键都将中断当前的活动。

附录 B 应答机的 SpecCharts 描述

```

-----
--Answering machine controller
-----

entity ansE is
  port (
    --Interface to line circuitry
    hangup_p      : in  bit; --hangup detected
    offhook_p     : out bit; --answers
    produce_beep_p : out bit; --produces a beep
    ring_p        : in  bit; --ring detected
    tone_p        : in  bit_vector (3 downto 0); --binary tone
    --Interface to display
    num_msgs_p    : out integer range 0 to 31; --msgs display
    on_light_p    : out bit; --turns on led
    --Touch--sensitive buttons
    but_fwd_p     : in  bit; --forward tape
    but_hear_ann_p : in  bit; --play pre--recorded announcement
    but_on_off_p  : in  bit; --toggle machine--on state
    but_memo_p    : in  bit; --record message via microphone
    but_play_msgs_p : in  bit; --play all messages
    but_play_p    : in  bit; --play tape from curr position
    but_rec_ann_p : in  bit; --record a new announcement
    but_rew_p     : in  bit; --rewind tape
    but_stop_p    : in  bit; --stop tape
    --Switches
    power_p       : in  bit; --power switch
    tollsaver_p   : in  bit; --answer after 2 rings if msg
    --Interface to announcement player
    ann_done_p    : in  bit; --end of announcement reached
    ann_play_p    : out bit; --plays announcement
    ann_rec_p     : out bit; --records announcement
    --Interface to tape player
    tape_fwd_p    : out bit; --forwards tape
    tape_play_p   : out bit; --plays tape
    tape_rew_p    : out bit; --rewinds tape
    tape_rec_p    : out bit; --records on tape
    tape_count_p  : in  integer --tape position, start is 0
  );
end;

architecture ansA of ansE is
begin

  behavior ans type concurrent substates is

```

```

--Global declarations
type four_buttons_type is array (1 to 4) of bit_vector(3 downto 0);
signal user_code      : four_buttons_type;    --user id number
signal machine_on     : bit; --current state of machine
signal machine_on_toggle : bit; --toggle on hangup
signal num_msgs       : integer range 0 to 31; --num of messages
signal any_button_pushed : bit; --1 if machine button pushed

begin

  Main : ;
  MachineOnToggler : ;
  ConcAsgns1 : ;
  ConcAsgns2 : ;

  --*****
  behavior Main type sequential substates is
  --*****
  --main control behavior of the answering machine controller
  --Terminates any tape player, announcement player, or beep
  --activity. Useful after exceptions such as hangup.
  --
  procedure TerminateAnyActivity is
  begin
    produce_beep_p <= '0';
    ann_rec_p <= '0';
    ann_play_p <= '0';
    tape_fwd_p <= '0';
    tape_play_p <= '0';
    tape_rew_p <= '0';
    tape_rec_p <= '0';
  end;

  --Produces a beep with the indicated length
  --
  procedure Beep (len : in time) is
  begin
    produce_beep_p <= '1';
    wait for len;
    produce_beep_p <= '0';
  end;

begin
  SystemOff : (TI, power_p = '1', SystemOn);
  SystemOn : (TI, power_p = '0', SystemOff);

  behavior SystemOff type code is
  begin

```

```

    TerminateAnyActivity;
    on_light.p <= '0'; --turn off led
end SystemOff;

```

behavior SystemOn type sequential substates is

```

signal terminal_tape_count: integer; --end of last message
signal toggle_on_hangup : bit; --toggle machine--on state

```

```

--Plays all messages by saving tape count of end of last
--message, rewinding to start of tape, and playing until
--end tape count.

```

```

procedure PlayAllMsgs

```

```

    (signal terminal_tape_count : inout integer ;
     signal tape_count : in integer ;
     signal tape_rew : out bit ;
     signal tape_play : out bit ) is

```

```

begin

```

```

    --Save tape count of end of last message
    terminal_tape_count <= tape_count.p;
    --Rewind to start of tape
    tape_rew <= '1';
    if not (tape_count = 0) then
        wait until tape_count = 0;
    end if;
    tape_rew <= '0';
    --Play until end of last message
    tape_play <= '1';
    if (tape_count < terminal_tape_count) then
        wait until tape_count = terminal_tape_count;
    end if;
    tape_play <= '0';
    --Beep to indicate that all messages have been played
    Beep(1 s);
end; --PlayAllMsgs

```

```

begin

```

```

    InitializeSystem :

```

```

        (TOC, true, RespondToLine);

```

```

    RespondToLine :

```

```

        (TI, any_button_pushed = '1'
         and any_button_pushed'event,
         RespondToMachineButton);

```

```

    RespondToMachineButton :

```

```

        (TOC, true, RespondToLine),
        (TI, any_button_pushed = '1'
         and any_button_pushed'event,
         RespondToMachineButton);

```

```

--Rewinds to beginning of tape, sets message number to 0
--
behavior InitializeSystem type code is
begin
    num_msgs <= 0;
    tape_rew_p <= '1';
    if (tape_count_p /= 0) then
        wait until tape_count_p = 0;
    end if;
    tape_rew_p <= '0';
    toggle_on_hangup <= '0';
end InitializeSystem;

```

```

--Monitor and answer line as opposed to handling
--machine buttons
--
behavior RespondToLine type sequential substates is
begin

```

```

    Monitor :
        (TOC, true, Answer),
        (TI, hangup_p = '1' and hangup_p'event, Monitor);
    Answer :
        (TOC, true, Monitor),
        (TI, machine_on = '0' and machine_on'event, Monitor);

```

```

--Monitors line for required number of rings
--If the machine is off, it answers after 15 rings
--(just in case the owner forgot to turn the machine
--on before leaving home).
--If the machine is on, it answers after 4 rings,
--UNLESS tollsaver is on and there is a message, in
--which case it answers after 2 rings.
--

```

```

behavior Monitor type code is

```

```

    variable rings_to_wait: integer range 1 to 20;
    variable i: integer range 0 to 20;

```

```

--Computes the number of rings to wait for
function DetermineRingsToWait return integer is
begin
    if ((num_msgs > 0) and (tollsaver_p = '1') and
        (machine_on = '1')) then
        return(2);
    elsif (machine_on = '1') then
        return(4);
    else

```

```

        return(15);
    end if;
end;

begin
    TerminateAnyActivity;
    --Turn on led if machine is on
    if (machine_on='1') then
        on_light_p <= '1';
    else
        on_light_p <= '0';
    end if;

    rings_to_wait := DetermineRingsToWait;
    i := 0;
    --Loop until required rings have been detected
    while (i < rings_to_wait) loop
        wait on tollsaver_p,machine_on,ring_p;
        if ring_p = '1' and ring_p'event then
            i := i + 1;
        end if;
        --If machine_on or tollsaver has changed, the
        --number of rings to wait may also change,
        --so let's recompute
        if (machine_on'event or tollsaver_p'event) then
            rings_to_wait := DetermineRingsToWait;
        end if;
    end loop ;
    offhook_p <= '1'; --answer the line
end Monitor;

--Answers the line.
--Normal sequence: PlayAnnouncement, RecordMsg, Hangup.
--If a hangup is detected while playing or recording,
--machine hangs up.
--If tone 1 is detected, enters remote operation mode.
--
behavior Answer type sequential substates is
begin

    PlayAnnouncement :
        (TI, tone_p = "0001", RemoteOperation),
        (TI, hangup_p = '1' and hangup_p'event, HANGUP),
        (TOC, true, RecordMsg);
    RecordMsg :
        (TI, tone_p = "0001", RemoteOperation),
        (TOC, true, Hangup);
    Hangup :
        (TOC, true, stop);

```

```

RemoteOperation :
    (TDC, true, HANGUP);

--Plays announcement until end of announcement
--
behavior PlayAnnouncement type code is
begin
    ann_play.p <= '1';
    wait until ann_done.p = '1';
    ann_play.p <= '0';
end PlayAnnouncement;

--Produces a beep, then records line until hangup or
--until a maximum time is reached.
--Places a beep at the end of the recorded message.
--
behavior RecordMsg type code is
begin
    Beep(1 s);
    tape_rec.p <= '1';
    if not (hangup.p = '1') then
        wait until hangup.p = '1' for 1000 s;
        Beep(1 s);
        num_msgs <= num_msgs + 1;
    end if;
    tape_rec.p <= '0';
end RecordMsg;

--Hangs up.    Toggles machine--on state if necessary
--
behavior Hangup type code is
begin
    offhook.p <= '0';
    ann_play.p <= '0';
    if (toggle_on_hangup = '1') then
        toggle_on_hangup <= '0';
        machine_on_toggle <= '1';
        wait for 1 s;
        machine_on_toggle <= '0';
    end if;
end Hangup;

--Processes remote commands given by machine owner if
--correct user identification number is entered
--
behavior RemoteOperation type sequential substates is
    signal code_ok : bit; --true if correct id
begin
    CheckUserId :

```

```

        (TOC, code_ok = '1', RespondToCmds),
        (TOC, code_ok = '0', stop),
        (TI, hangup_p = '1', stop);
RespondToCmds :
    (TOC, true, stop);

--Checks next four button--tones against user id,
--sets code_ok to true if all four match
behavior CheckUserId type code is
    variable entered_code: four_buttons_type;
    variable i: integer range 1 to 5;
begin
    TerminateAnyActivity;
    code_ok <= '1';
    i := 1;
    while i <= 4 loop
        wait until tone_p /= "1111" and tone_p'event;
        if (tone_p /= user_code(i)) then --wrong
            code_ok <= '0';
        end if;
        i := i + 1;
    end loop ;
end CheckUserId;

```

```

--Processes user commands.    When done with
--commands, resets tape to end of last message,
--unless of course the user has erased all
--messages.    HearMsgsCmds is the initial mode
--which allows commands related simply to hearing
--messages.    If tone="0010" is detected, enters
--MiscCmds, in which miscellaneous, more advanced
--commands related to machine maintenance
--can be applied.
--

```

```

behavior RespondToCmds type sequential substates
is begin

```

```

    HearMsgsCmds :
        (TOC, true, MiscCmds),
        (TI, hangup_p = '1', ResetTape);
    MiscCmds :
        (TOC, tone_p = "0010", HearMsgsCmds),
        (TOC, other, ResetTape),
        (TI, hangup_p = '1', ResetTape);
    ResetTape :
        (TOC, true, stop);

```

```

--Normal command processing mode.    All comman
--related to hearing messages can be applied.
--

```

```

behavior HearMsgsCmds type code is
  variable i : integer;
begin
  if (tone_p = "1111") then
    wait until tone_p /= "1111";
  end if;

  tape_play_p <= '0';
  tape_fwd_p <= '0';
  tape_rew_p <= '0';
  --"1000" enters MiscCmds
  if (tone_p /= "1000") then
    case (tone_p) is
      when "0010" => --play all messages
        PlayAllMsgs(terminal_tape_count,
                    tape_count_p,
                    tape_rew_p, tape_play_p);
      when "0011" => --play tape
        tape_play_p <= '1';
      when "0100" => --forward tape
        tape_fwd_p <= '1';
      when "0101" => --rewind tape to start
        tape_rew_p <= '1';
        if (tape_count_p /= 0) then
          wait until tape_count_p = 0;
        end if;
        tape_rew_p <= '0';
      when "0110" => --stop tape
        tape_play_p <= '0';
        tape_fwd_p <= '0';
        tape_rew_p <= '0';
      when "0111" => --beep number messages
        wait for 5 s;
        i := 0;
        --one beep / msg
        while (i < num_msgs) loop
          Beep(1 s);
          wait for 1 s;
          i := i + 1;
        end loop ;
      when others =>
    end case;
  end if;
end HearMsgsCmds;

--In this mode the user can perform less
--common commands related to machine
--maintenance.

```

```

---
behavior MiscCmds type code is
begin
  --Indicate new mode with a beep
  Beep(1 s);
  loop
    wait until tone_p /= "1111"
      and tone_p'event;
    case (tone_p) is
      when "0010" => --exit MiscCmds mode
        exit; --exit loop
      when "0011" => --rewind tape
        tape_rew_p <= '1';
        if not (tape_count_p = 0) then
          wait until tape_count_p = 0;
        end if;
        tape_rew_p <= '0';
        terminal_tape_count <= 0;
      when "0100" => --hear announcement
        ann_play_p <= '1';
        wait until ann_done_p = '1';
        ann_play_p <= '0';
      when "0101" => --record announcement
        --preparation time
        wait for 50 s;
        --beep indicates start
        Beep(1 s);
        wait for 0 s;
        --record for full length
        ann_rec_p <= '1';
        wait until ann_done_p = '1';
        ann_rec_p <= '0';
        --beep indicates end
        Beep(1 s);
      when "0110" => --toggle mach--on stat
        toggle_on_hangup <= '1';
      when others =>
    end case;
  end loop;
end MiscCmds;

```

--Reset tape to end of last message.

--Rewinds if past end, forwards if before end.

```

behavior ResetTape type code is
  variable tape_count: integer;

```

```

begin
  if (tape_count_p > terminal_tape_count) then
    tape_rew_p <= '1';
    wait until
      (tape_count_p <= terminal_tape_count);

```

```

        tape_rew_p <= '0';
    elseif (tape_count_p < terminal_tape_count) then
        tape_fwd_p <= '1';
        wait until
            (tape_count_p >= terminal_tape_count);
        tape_fwd_p <= '0';
    end if;
end ResetTape;
end RespondToCmds;
    end RemoteOperation;
    end Answer;
end RespondToLine;

--Processes command indicated by a button being pressed
--on the machine.
behavior RespondToMachineButton type code is

```

```

    procedure HandlePlayPushed is
    begin
        tape_play_p <= '1';
        num_msgs <= 0;
    end ;

```

```

    procedure HandleFwdPushed is
    begin
        tape_fwd_p <= '1';
        num_msgs <= 0;
    end ;

```

```

    procedure HandleRewPushed is
    begin
        num_msgs <= 0;
        tape_rew_p <= '1';
        if (tape_count_p /= 0) then
            wait until tape_count_p = 0;
        end if;
        tape_rew_p <= '0';
    end ;

```

```

    procedure HandleMemoPushed is --record message via mic.
    begin
        Beep(1 s);
        tape_rec_p <= '1';
        wait until but_memo_p = '0' for 1000 s;
        Beep(1 s);
        num_msgs <= num_msgs + 1;
        tape_rec_p <= '0';
    end ;

```

```

    procedure HandleStopPushed is

```

```

begin
    num_msgs <= 0;
    tape_play_p <= '0';
    tape_fwd_p <= '0';
    tape_rew_p <= '0';
    tape_rec_p <= '0';
end ;

procedure HandleHearAnnPushed is --play announcement
begin
    ann_play_p <= '1';
    wait until ann_done_p = '1';
    ann_play_p <= '0';
end ;

procedure HandleRecAnnPushed is --record announcement
begin
    wait for 50 s;
    Beep(1 s);
    wait for 0 s;
    ann_rec_p <= '1';
    wait until ann_done_p = '1';
    ann_rec_p <= '0';
    Beep(1 s);
end ;

procedure HandlePlayMsgsPushed is --play all msgs
begin
    terminal_tape_count <= tape_count_p;
    tape_rew_p <= '1';
    if not (tape_count_p = 0) then
        wait until tape_count_p = 0;
    end if;
    tape_rew_p <= '0';
    tape_play_p <= '1';
    if (tape_count_p < terminal_tape_count) then
        wait until tape_count_p = terminal_tape_count;
    end if;
    tape_play_p <= '0';
end ;

```

```

begin --RespondToMachineButton

```

```

    if (but_play_p='1') then
        HandlePlayPushed;
    elsif (but_fwd_p='1') then
        HandleFwdPushed;
    elsif (but_rew_p='1') then
        HandleRewPushed;

```

```

        elsif (but_memo_p='1') then
            HandleMemoPushed;
        elsif (but_stop_p='1') then
            HandleStopPushed;
        elsif (but_hear_ann_p='1') then
            HandleHearAnnPushed;
        elsif (but_rec_ann_p='1') then
            HandleRecAnnPushed;
        elsif (but_play_msgs_p='1') then
            HandlePlayMsgsPushed;
        end if;

        end RespondToMachineButton;
    end SystemOn;

end Main;

behavior MachineOnToggler type code is
begin
    machine_on <= '0';
    loop
        wait until but_on_off_p = '1' or machine_on_toggle = '1';
        if machine_on = '0' then
            machine_on <= '1';
        else
            machine_on <= '0';
        end if;
    end loop ;
end MachineOnToggler;

--Sets any_button_pushed to 1 if any machine button is pushed.
--
behavior ConcAsgns1 type code is
begin
    loop
        wait on but_play_p, but_fwd_p, but_rew_p, but_memo_p, but_stop_p,
            but_hear_ann_p, but_rec_ann_p, but_play_msgs_p;
        if (but_play_p = '1' and but_play_p'event) or
            (but_fwd_p = '1' and but_fwd_p'event) or
            (but_rew_p = '1' and but_rew_p'event) or
            (but_memo_p = '1' and but_memo_p'event) or
            (but_stop_p = '1' and but_stop_p'event) or
            (but_hear_ann_p = '1' and but_hear_ann_p'event) or
            (but_rec_ann_p = '1' and but_rec_ann_p'event) or
            (but_play_msgs_p = '1' and but_play_msgs_p'event) then
            any_button_pushed <= '1';
        else
            any_button_pushed <= '0';
        end if;
    end loop ;

```

```
end ConcAsgns1;

--Updates num_msgs_p port with current value of internal signal
--num_msgs so that numerical display shows number of messages.
--
behavior ConcAsgns2 type code is
begin
    loop
        wait on num_msgs;
        if (num_msgs'event) then
            num_msgs_p <= num_msgs;
        end if;
    end loop ;
end ConcAsgns2;

end ans;
end ansA;
```


参考文献

- [AB91] T. Amon and G. Borriello. "Sizing synchronization queues: A case study in higher level synthesis". In *Proceedings of the Design Automation Conference*, 1991.
- [Ake91] J. Akella. *Input/Output Performance Modeling and Interface Synthesis in Concurrently Communicating Systems*. PhD thesis, Carnegie Mellon University, November 1991.
- [AM91] J. Akella and K. McMillan. "Synthesizing converters between finite state protocols". In *Proceedings of the International Conference on Computer Design*, 1991.
- [AS83] G.R. Andrews and F. Schneider. "Concepts and notations for concurrent programming". *ACM Computing Surveys*, 15(1): 3-44, March 1983.
- [ASU88] A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1988.
- [Aul91] R.J. Auletta. "CSP specified digital repeater and translation to synchronous logic". Technical Report, Dept. ECE, George Mason University, 1991.
- [AWC90] A. Arsenault, J.J. Wong, and M. Cohen. "VHDL transition from system to detailed design". In *VHDL Users' Group*, April 1990.
- [Bar81] M.R. Barbacci. "Instruction set processor specifications (ISPS): The notation and its applications". In *IEEE Transactions on Computers*, January, 1981.
- [BCD⁺88] R.K. Brayton, R. Camposano, G. DeMicheli, R.H. Otten, and J.T.J. van Eijndhoven. "The Yorktown silicon compiler system". in D.D. Gajski, Editor, *Silicon Compilation*, Addison-Wesley, 1988.
- [BE89] O.A. Buset and M.I. Elmasry. "ACE: A hierarchical graphical interface for architectural synthesis". In *Proceedings of the Design Automation Conference*, pages 537-542, 1989.
- [Ber91] G. Berry. "A hardware implementation of pure Esterel". Digital Equipment Paris Research Laboratory, July, 1991.

- [BHS91] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specifications*. Prentice Hall, 1991.
- [BK87] G. Borriello and R.H. Katz. "Synthesis and optimization of interface transducer logic". In *Proceedings of the International Conference on Computer-Aided Design*, 1987.
- [BM89] M. Balakrishnan and P. Marwedel. "Integrated scheduling and binding: A synthesis approach for design space exploration". In *Proceedings of the Design Automation Conference*, 1989.
- [Boo91] G. Booch. *Object-oriented Design with Applications*. Benjamin/Cummings, Redwood City, California, 1991.
- [Bor88] G. Borriello. *A New Interface Specification Methodology and its Applications to Transducer Synthesis*. PhD thesis, University of California, Berkeley, May 1988.
- [Bor91] G. Borriello. "Specification and synthesis of interface logic". In R. Camposano and W. Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1991.
- [BRSVW87] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. "MIS: A multiple-level logic optimization system". *IEEE Transactions on Computer-Aided Design*, 6(6): 1062-1080, November 1987.
- [CB87] R. Camposano and R.K. Brayton. "Partitioning before logic synthesis". In *Proceedings of the International Conference on Computer-Aided Design*, 1987.
- [CG93] V. Chaiyakul and D.D. Gajski. "High-level transformations for minimizing syntactic variances". In *Proceedings of the Design Automation Conference*, 1993.
- [Che77] P. S. Chen. *The Entity-Relationship Approach to Logical Data Base Design*. Q.E.D. Information Sciences, Wellesley, Massachusetts, 1977.
- [CLR89] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1989.
- [CPTR89] C.M. Chu, M. Potkonjak, M. Thaler, and J. Rabaey. "HYPER: An interactive synthesis environment for high perfor-

- mance real time applications". In *Proceedings of the International Conference on Computer Design*, pages 432-435, 1989.
- [CR89] R. Camposano and W. Rosenstiel. "Synthesizing circuits from behavioral descriptions". *IEEE Transactions on Computer-Aided Design*, 8(2): 171-180, February 1989.
- [CS86] C.Tseng and D.P. Siewiorek. "Automated synthesis of datapaths in digital systems". *IEEE Transactions on Computer-Aided Design*, pages 379-395, July 1986.
- [CST91] R. Camposano, L.F. Saunders, and R.M. Tabet. "High-level synthesis from VHDL". In *IEEE Design & Test of Computers*, 1991.
- [CT89] R. Camposano and R.M. Tabet. "Design representation for the synthesis of behavioral VHDL models". In *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications*, 1989.
- [CT90] R. Cloutier and D.E. Thomas. "The combination of scheduling, allocation and mapping in a single algorithm". In *Proceedings of the Design Automation Conference*, June 1990.
- [CvE87] R. Camposano and J.T. van Eijndhoven. "Partitioning a design in structural synthesis". In *Proceedings of the International Conference on Computer Design*, 1987.
- [Dav83] W. S. Davis. *Tools and Techniques for Structured Systems Analysis and Design*. Addison-Wesley, Reading, Massachusetts, 1983.
- [DCH91] N. Dutt, J. Cho, and T. Hadley. "A user interface for VHDL behavioral modeling". In *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications*, 1991.
- [DeM79] T. DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1979.
- [DH89] D. Drusinsky and D. Harel. "Using Statecharts for hardware description and synthesis". In *IEEE Transactions on Computer-Aided Design*, 1989.
- [DK88] G. DeMicheli and D.C. Ku. "HERCULES - a system for

- high-level synthesis". In *Proceedings of the Design Automation Conference*, 1988.
- [DMNSV88] S. Devadas, H.K.T. Ma, A.R. Newton, and A. Sangiovanni-Vincentelli. "MUSTANG: State assignment of finite state machines targeting multilevel logic implementations". *IEEE Transactions on Computer-Aided Design*, 7(12): 1290-1299, 1988.
- [DSVA87] G. DeMicheli, A. Sangiovanni-Vincentelli, and P. Antognetti. *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*. Martinus Nijhoff Publishers, 1987.
- [EHB94] R. Ernst, J. Henkel, and T. Benner. "Hardware-software cosynthesis for microcontrollers". In *IEEE Design & Test of Computers*, pages 64-75, December 1994.
- [EST78] D.P. Siewiorek E.A. Snow and D.E. Thomas. "A technology-relative computer-aided design system: Abstract representation, transformations and design trade-offs". In *Proceedings of the Design Automation Conference*, 1978.
- [FKCD93] D. Filo, D. Ku, C.N. Coelho, and G. DeMicheli. "Interface optimization for concurrent systems under timing constraints". In *IEEE Transactions on Very Large Scale Integration Systems*, pages 268-281, September 1993.
- [FM82] C.M. Fiduccia and R.M. Mattheyses. "A linear-time heuristic for improving network partitions". In *Proceedings of the Design Automation Conference*, 1982.
- [Gaj91] D.D. Gajski. "Essential issues and possible solutions in high-level synthesis". in R. Camposano and W. Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1991.
- [GD90] R. Gupta and G. DeMicheli. "Partitioning of functional models of synchronous digital systems". In *Proceedings of the International Conference on Computer-Aided Design*, pages 216-219, 1990.
- [GD92] R. Gupta and G. DeMicheli. "System-level synthesis using re-programmable components". In *Proceedings of the European Conference on Design Automation (EDAC)*, pages 2-7, 1992.

- [GD93] R. Gupta and G. DeMicheli. "Hardware-software cosynthesis for digital systems". In *IEEE Design & Test of Computers*, pages 29–41, October 1993.
- [GDWL91] D.D. Gajski, N.D. Dutt, C.H. Wu, and Y.L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, Massachusetts, 1991.
- [Geb92a] C.H. Gebotys. "Optimal scheduling and allocation of embedded VLSI chips". In *Proceedings of the Design Automation Conference*, pages 116–119, 1992.
- [Geb92b] C.H. Gebotys. "Optimal synthesis of multichip architectures". In *Proceedings of the International Conference on Computer-Aided Design*, pages 238–241, 1992.
- [GGN94] J. Gong, D.D. Gajski, and S. Narayan. "Software estimation from executable specifications". In *Journal of Computer and Software Engineering*, 1994.
- [Gif78] W. Giffin. *Queueing: Basic Theory and Applications*. Grid Inc., Columbus, Ohio, 1978.
- [GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, 1979.
- [GSV92] A. Gutierrez, P. Sanchez, and E. Villar. "VHDL high-level silicon compilation: Synthesis methodology and teaching experience". In *Proceedings of the Third Eurochip Workshop on VLSI Design Training*, 1992.
- [GVN93] D.D. Gajski, F. Vahid, and S. Narayan. "SpecCharts: A VHDL front-end for embedded systems". UC Irvine, Dept. of ICS, Technical Report 93-31, 1993.
- [GVN94] D.D. Gajski, F. Vahid, and S. Narayan. "A system-design methodology: Executable-specification refinement". In *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.
- [Hal93] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.
- [Har87] D. Harel. "Statecharts: A visual formalism for complex systems". *Science of Computer Programming* 8, 1987.
- [HAWW88] F.T. Hady, J.H. Aylor, R.D. Williams, and R. Waxman.

- "Uninterpreted modeling using the VHSIC hardware description language (VHDL)". In *Proceedings of the International Conference on Computer-Aided Design*, pages 172-175, 1988.
- [HB85] K. Hwang and F. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1985.
- [HG91] T. Hadley and D.D. Gajski. "A decision support environment for behavioral synthesis". UC Irvine, Dept. of ICS, Technical Report 91-17, 1991.
- [Hil85] P. Hilfinger. "A high-level language and silicon compiler for digital signal processing". In *Proceedings of the Custom Integrated Circuits Conference*, 1985.
- [HLN⁺88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring. "STATEMATE: A working environment for the development of complex reactive systems". In *Proceedings of the International Conference on Software Engineering*, 1988.
- [Hoa78] C.A.R. Hoare. "Communicating sequential processes". *Communications of the ACM*, 21(8): 666-677, 1978.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.
- [HP90] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quatitative Approach*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.
- [HR92] P. Hilfinger and J. Rabey. *Anatomy of a Silicon Compiler*. Kluwer Academic Publishers, 1992.
- [HRSV86] M.D. Hung, F. Romeo, and A. Sangiovanni-Vincentelli. "An efficient general cooling schedule for simulated annealing". In *Proceedings of the International Conference on Computer-Aided Design*, pages 381-384, 1986.
- [HS71] A. Hashimoto and J. Stevens. "Wire routing by optimizing channel assignments within large apertures". In *Proceedings of the Design Automation Conference*, 1971.
- [HT93] J.W. Hagerman and D.E. Thomas. "Process transformation for system level synthesis". Technical Report

- CMUCAD-93-08, 1993.
- [IEE88] IEEE Inc., N.Y. *IEEE Standard VHDL Language Reference Manual*, 1988.
- [IOJ94] T.B. Ismail, K O'Brien, and A.A. Jerraya. "Interactive system-level partitioning with Partif". In *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.
- [JMP88] R. Jain, M. Mlinar, and A. Parker. "Area-time model for synthesis of non-pipelined designs". In *Proceedings of the International Conference on Computer-Aided Design*, 1988.
- [Joh67] S.C. Johnson. "Hierarchical clustering schemes". *Psychometrika*, pages 241-254, September 1967.
- [JPA91] A. Jerraya, P. Paulin, and D. Agnew. "Facilities for controllers modeling and synthesis in VHDL". In *VHDL Users' Group*, April 1991.
- [KC91] Y.C. Kirkpatrick and C.K. Cheng. "Ratio cut partitioning for hierarchical designs". *IEEE Transactions on Computer-Aided Design*, 10(7): 911-921, 1991.
- [KD88] D.C. Ku and G. DeMicheli. "HardwareC - a language for hardware design". Stanford University, Technical Report CSL-TR-90-419, 1988.
- [KD91] D. Ku and G. DeMicheli. "Synthesis of ASICs with Hercules and Hebe". in R. Camposano and W. Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1991.
- [KGRC93] F.J. Kurdahi, D.D. Gajski, C. Ramachandran, and V. Chaiyakul. "Linking register-transfer in physical levels of design". In *IEICE Transactions on Information and Systems, Vol E76-D, No 9*, September 1993.
- [KGV83] S. Kirkpatrick, C.D. Gelatt, and M. P. Vecchi. "Optimization by simulated annealing". *Science*, 220(4598): 671-680, 1983.
- [KL70] B.W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs". *Bell System Technical Journal*, February 1970.
- [KL93] A. Kalavade and E.A. Lee. "A hardware/software codesign methodology for DSP applications". In *IEEE Design & Test of Computers*, 1993.

- [KP87] F.J. Kurdahi and A.C. Parker. "Real: A program for register allocation". In *Proceedings of the Design Automation Conference*, 1987.
- [KP91] K. Kucukcakar and A. Parker. "CHOP: A constraint-driven system-level partitioner". In *Proceedings of the Design Automation Conference*, 1991.
- [KR78] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.
- [Kri84] B. Krishnamurthy. "An improved min-cut algorithm for partitioning VLSI networks". *IEEE Transactions on Computers*, May 1984.
- [KV93] N. Kumar and R. Vemuri. "Partitioning for multicomponent synthesis from VHDL specifications". In *VHDL International Users' Forum*, pages 19–28, 1993.
- [KWK85] S. Kung, H. Whitehouse, and T. Kailath. *VLSI and Modern Signal Processing*. Prentice-Hall, 1985.
- [Lag89] E.D. Lagnese. *Architectural Partitioning for System Level Design of Integrated Circuits*. PhD thesis, Carnegie Mellon University., March 1989.
- [Laz84] E. D. Lazowska. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.
- [Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley and Sons, England, 1990.
- [LG88] J. Lis and D.D. Gajski. "Synthesis from VHDL". In *Proceedings of the International Conference on Computer Design*, 1988.
- [LGP+91] D. Lanneer, G. Goossens, M. Pauwels, J. Van Meerbergen, and H. De Man. "An object-oriented framework supporting the full high-level synthesis trajectory". In *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications*, pages 281–300, 1991.
- [LGR92] B. Lutter, W. Glunz, and F.J. Rammig. "Using VHDL for simulation of SDL specifications". In *Proceedings of the European Design Automation Conference (EuroDAC)*, pages 630–635, 1992.

- [LHHR92] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. "Requirements specification for process-control systems". UC Irvine, Dept. of ICS, Technical Report 92-106, 1992.
- [Lis92] J. Lis. *Behavioral Synthesis from VHDL Using Structured Modeling*. PhD thesis, University of California, Irvine, January 1992.
- [LND⁺91] D. Lanneer, S. Note, F. Depuydt, M. Pauwels, F. Catthoor, G. Goosens, and H. De Man. "Architectural synthesis for medium and high throughput signal processing with the new CATHEDRAL environment". In R. Camposano and W. Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1991.
- [LT89] E.D. Lagnese and D.E. Thomas. "Architectural partitioning for system level design". In *Proceedings of the Design Automation Conference*, 1989.
- [LT91] E.D. Lagnese and D.E. Thomas. "Architectural partitioning for system level synthesis of integrated circuits". *IEEE Transactions on Computer-Aided Design*, July 1991.
- [MAP93] P. Moeschler, H.P. Amann, and F. Pellandini. "High-level modeling using extended timing diagrams". In *Proceedings of the European Design Automation Conference (EuroDAC)*, 1993.
- [Mar91] F. Maraninchi. "Argos: A graphical synchronous language, for the description of reactive systems". Report RT-C29, Univeristy Joseph Fourier, 1991.
- [McF86] M.C. McFarland. "Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions". In *Proceedings of the Design Automation Conference*, 1986.
- [MK90] M.C. McFarland and T.J. Kowalski. "Incorporating bottom-up design into hardware synthesis" *IEEE Transactions on Computer-Aided Design*, September 1990.
- [MW90] R. MacDonald and R. Waxman. "Operational specification of the SINGARS radio in VHDL". In *AFCEA-IEEE Tactical Communications Conference*, pages 1-17, 1990.
- [Nes87] J.A. Nestor. *Specification and Synthesis of Digital System*

- with Interfaces*. PhD thesis, Carnegie Mellon University, April 1987.
- [NG92] S. Narayan and D.D. Gajski. "System clock estimation based on clock slack minimization". In *Proceedings of the European Design Automation Conference (EuroDAC)*, 1992.
- [NG93] S. Narayan and D.D. Gajski. "Features supporting system specification in HDLs". In *Proceedings of the European Design Automation Conference (EuroDAC)*, 1993.
- [NG94] S. Narayan and D.D. Gajski. "Synthesis of system-level bus interfaces". In *Proceedings of the European Conference on Design Automation (EDAC)*, 1994.
- [NON91] Y. Nakamura, K. Oguri, and A. Nagoya. "Synthesis from pure behavioral descriptions". In R. Camposano and W. Wolf, Editors, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1991.
- [NT86] J. Nestor and D. Thomas. "Behavioral synthesis with interfaces". In *Proceedings of the International Conference on Computer-Aided Design*, p.112-115, 1986.
- [NVG91a] S. Narayan, F. Vahid, and D.D. Gajski. "System specification and synthesis with the SpecCharts language". In *Proceedings of the International Conference on Computer-Aided Design*, 1991.
- [NVG91b] S. Narayan, F. Vahid, and D.D. Gajski. "Translating system specifications to VHDL". In *Proceedings of the European Conference on Design Automation (EDAC)*, 1991.
- [NVG92] S. Narayan, F. Vahid, and D.D. Gajski. "System specification with the SpecCharts language". In *IEEE Design & Test of Computers*, Dec. 1992.
- [OG86] A. Orailoglu and D.D. Gajski. "Flow graph representation". In *Proceedings of the Design Automation Conference*, 1986.
- [OvG84] R. Otten and L. van Ginneken. "Floorplan design using simulated annealing". In *Proceedings of the International Conference on Computer-Aided Design*, p.96-98, 1984.
- [Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Sys-*

- tems. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
- [PF89] D. Pang and L. Ferrari. "Unified approach to general IFIR filter design using the B-spline function". In *Proceedings of Asilomar Conference on Signals, Systems & Computers*, 1989.
- [PK89a] P.G. Paulin and J.P. Knight. "Algorithms for high-level synthesis". In *IEEE Design & Test of Computers*, Dec. 1989.
- [PK89b] P.G. Paulin and J.P. Knight. "Force-directed scheduling for the behavioral synthesis of ASICs". *IEEE Transactions on Computer-Aided Design*, June 1989.
- [PKG86] P.G. Paulin, J.P. Knight, and E.F. Girzyc. "HAL: A multi-paradigm approach to datapath synthesis". In *Proceedings of the Design Automation Conference*, 1986.
- [PP85] N. Park and A.C. Parker. "Synthesis of optimal clocking schemes". In *Proceedings of the Design Automation Conference*, 1985.
- [PPM86] A.C. Parker, T. Pizzaro, and M. Mlinar. "MAHA: A program for datapath synthesis". In *Proceedings of the Design Automation Conference*, 1986.
- [Rei92] W. Reisig. *A Primer in Petri Net Design*. Springer-Verlag, New York, 1992.
- [RG91] E.A. Rundensteiner and D.D. Gajski. "A design data base for behavioral synthesis". In *Proceedings of the International Workshop on High-Level Synthesis*, 1991.
- [RG93] L. Ramachandran and D.D. Gajski. "Architectural trade-offs in synthesis of pipelined controls". *Proceedings of the European Design Automation Conference (EuroDAC)*, 1993.
- [RSV85] F. Romeo and A. Sangiovanni-Vincentelli. "Probabilistic hill climbing algorithms: Properties and applications". In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, p. 393-417, 1985.
- [RVNG92] L. Ramachandran, F. Vahid, S. Narayan, and D.D. Gajski. "Semantics and synthesis of signals in behavioral VHDL". In *Proceedings of the European Design Automation Confer-*

- ence (*EuroDAC*), 1992.
- [SB92] J.S. Sun and R.W. Brodersen. "Design of system interface modules". In *Proceedings of the International Conference on Computer-Aided Design*, pages 478–481, 1992.
- [Sod90] Jag Sodhi. *Computer Systems Techniques: Development, Implementation and Software Maintenance*. TAB Professional and Reference Books, Blue Ridge Summit, Pennsylvania, 1990.
- [SP91] S.Prakash and A.C. Parker. "Synthesis of application-specific multiprocessor architectures". In *Proceedings of the Design Automation Conference*, p. 8–13, 1991.
- [SSB91] J.S. Sun, M.B. Srivastava, and R.W. Brodersen. "SIERA: A CAD environment for real-time systems". In *3rd Physical Design Workshop*, May 1991.
- [SST90] E. Sternheim, R. Singh, and Y. Trivedi. *Hardware Modeling with Verilog HDL*. Automata Publishing Company, Cupertino, CA, 1990.
- [Sut88] A. Sutcliffe. *Jackson System Development*. Prentice-Hall, New York, 1988.
- [TAS93] D.E. Thomas, J.K. Adams, and H. Schmit. "A model and methodology for hardware/software codesign". In *IEEE Design & Test of Computers*, p. 6–15, 1993.
- [Teo90] T. J. Teorey. *Database Modeling and Design: The Entity-relationship Approach*. Morgan Kaufman Publishers, San Mateo, California, 1990.
- [TLK90] T. Tikanen, T. Leppanen, and J. Kivela. "Structured analysis and VHDL in embedded ASIC design and verification". In *Proceedings of the European Conference on Design Automation (EDAC)*, p. 107–111, 1990.
- [TLW⁺90] D.E. Thomas, E.D. Langese, R.A. Walker, J.A. Nestor, J.V. Rajan, and R.L. Blackburn. *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*. Kluwer Academic Publishers, 1990.
- [TM91] D.E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.
- [VG91] F. Vahid and D.D. Gajski. "Obtaining functionally equiva-

- lent simulations using VHDL and a time-shift transformation". In *Proceedings of the International Conference on Computer-Aided Design*, 1991.
- [VG92] F. Vahid and D.D. Gajski. "Specification partitioning for system design". In *Proceedings of the Design Automation Conference*, 1992.
- [VNG91] F. Vahid, S. Narayan, and D.D. Gajski. "SpecCharts: A language for system level synthesis". In *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications*, 1991.
- [VTI88] *VDP100 1.5 Micron CMOS Datapath Cell Library*, 1988.
- [WC91] R. Walker and R. Camposano. *A Survey of High-Level Synthesis Systems*. Kluwer Academic Publishers, 1991.
- [WCG91] C.H. Wu, V. Chaiyakul, and D.D. Gajski. "Layout-area models for high-level synthesis". In *Proceedings of the International Conference on Computer-Aided Design*, 1991.
- [WM85] P. T. Ward and S. J. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, New York, 1985.
- [WT89] R.A. Walker and D.E. Thomas. "Behavioral transformation for algorithmic level IC design". *IEEE Transactions on Computer-Aided Design*, October 1989.
- [Xil89] Xilinx Corporation. *The Programmable Gate Array Data Book*, 1989.
- [YC78] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press, New York, 1978.
- [YEBH93] W. Ye, R. Ernst, T. Benner, and J. Henkel. "Fast timing analysis for hardware-software co-synthesis". In *Proceedings of the International Conference on Computer Design*, p. 452-457, 1993.
- [YH90] H.S. Yoo and A. Hsu. "Debbie: A configurable user interface for CAD frameworks". In *Proceedings of the International Conference on Computer Design*, p. 135-140, 1990.
- [Zim88] G. Zimmerman. "A new area and shape function estimation technique for VLSI layouts". In *Proceedings of the Design Automation Conference*, 1988.

术语解释

abstraction level (抽象级别)——概念模型与系统描述中的详细实现程度。实现越详细,抽象级别越低。

allocation (分配)——(动词)在设计中增加一个新系统组件(譬如一个 ASIC)的动作。(名词)设计中的系统组件的集合。(术语“分配”在高层次综合领域中也很常用,表示在一个设计中增加一个寄存器级元件。)

Application-Specific Integrated Circuit (ASIC)——(专用集成电路)实现数字功能的全定制设计的芯片。

behavior (行为)——一部分系统功能,比一个算术运算更复杂。

capture-and-simulate (捕获-模拟)——一种设计方法,首先实现系统,再对其进行信息捕获,形成寄存器传输级或门级网表,然后对其模拟,验证功能是否正确。

channel (通道)——在两个并发生行为之间的抽象通信媒介,在其中进行数据传输。

closeness (接近性)——以一个或多个度量形式表示的划分过程中两个对象结群的期望值数。

component (组件)——实现功能的一个物理对象。

constraint (约束)——给定度量的最大或最小值。

cost (成本)——一个目标函数的返回值,表示给定设计的期望值。参见 *monetary cost*。

describe-and-synthesize (描述-综合)——一种设计方法,首先描述系统,作为可执行规范化描述,经过模拟验证其功能的正确性,然后对其进行实现,或者用手工,或者用综合工具。

description (描述)——参见 *specification*。

design (设计)——(名词)实现系统的部分功能或所有功能的结构。(动词)在系统中增加结构的活动。

designer (设计者)——进行系统设计的人。

estimation (评估)——其功能尚未完全实现时,估计其设计的计算度量值,也许只是部分实现。

executable language (可执行语言)——计算机可读的且可模拟的语言。

functional object (功能对象)——一个可实现的系统规范描述中的变量,行为或者通道。

functional partitioning (功能划分)——划分系统组件中功能对象的动作。

implement (实现)——创建执行给定功能的结构的动作。有时也指执行一个给定功能的结构的动作,例如,用 ALU 实现算术运算。

implementation (实现)——在适合于制造的一个详细的级别完全实现一个系统的功能的结构。

interface process (接口进程)——在不相容的通信行为的协议之间完成数据传输的一个行为。

high-level synthesis (高层次综合)——把一个 ASIC 可执行规范描述转换到寄存器传输级结构的动作,包括 3 个任务:调度、分配和绑定。这些任务可以部分或完全自动实现。

metric (度量)——用来评价一个实现的质量或期望值的参数。

modeler (建模者)——其任务为编写可执行规范描述的人。

monetary cost (费用)——一个实现的成本价格。

natural language (自然语言)——口语语言。

objective function(目标函数)——评价一个设计的期望值的函数,基于质量的度量和约束。

partition(划分)——(名词)把对象分成若干组。

partitioning(划分)——(动词)把各个对象由对象集合分成若干组的动作。

process(进程)——与其他行为并发执行的行为。

processor(处理器)——通过状态集合顺序执行的实现功能的系统组件,其中每个状态中设置控制线的值,控制数据通路的数据传播,从而实现所要求的计算。该组件可以是标准的,可购买现货的处理器,如 Intel 8086,也可以是定制设计的。

protocol(协议)——详细指定数据传输顺序和控制一个固定连线集合上的信号流的通信。

RT component(寄存器传输级元件)——实现寄存器传输级功能的物理对象,如 ALU、计数器或寄存器。

selection(选择)——在设计中增加一个新的寄存器传输级元件,如 ALU。

specification(系统描述)——对所期望的系统功能编写的描述。自然语言系统描述用人们的口头语言来描述。可执行系统描述用形式化的、机器可读的语言描述,它可以被模拟,以便验证其功能。

standard component(标准元件)——预先设计的市场上出售的元件,有固定的功能和接口。

standard language(标准语言)——一种可执行的语言,通用,并有工具支持。

structural partitioning(结构划分)——将系统的组件划分为结构组件的动作。

structure(结构)——组件的互联。

system(系统)——包括输入和输出的集合和功能描述,或者再加上实现功能的组件。功能描述由一个或多个算法组成,比算术运算更复杂。

system component(系统组件)——实现系统级功能的物理对象,如 ASIC、处理器或存储器。

system design(系统设计)——一种活动:分配系统组件,在这些组件中划分系统功能,并定义每个组件的功能,以便实现系统功能。

system functionality(系统功能)——系统完整的作用,系统的输出定义为其输入和时间的函数。

索引

索引中的页码为英文原书页码,与书中边栏标注的页码一致。

A

- Abstraction levels(抽象级别), 1, 4~6, 63, 66, 71
 - gate(门级), 5, 7
 - processor(处理器级), 5
 - register(寄存器级), 5, 7
 - structure(结构级), 178
 - tasks(任务级), 178
 - transistor(晶体管级), 4
- Allocation(分配), 8, 12, 171, 180, 225, 381, 387
- Allocation, functional-units, *see* Functional-unit selection
(功能单元分配, 参见 functional-unit selection)
- Answering machine(应答机), 118, 138, 140
- Aparty(一种设计系统), 209, 300
- Arbitration(仲裁)
 - models(仲裁模型), 330
 - process generation(仲裁进程生成), 333
 - schemes(仲裁模式), 332
- Architecture(体系结构), 18, 19
 - application-specific architecture(专用体系结构), 47
 - controller(控制器), 47
 - datapath(数据通路), 48
 - FSMD(带数据通路的有限状态机), 50
 - general-purpose processor(通用处理器), 51
 - CISC(复杂指令集计算机), 51
 - RISC(精简指令集计算机), 53
 - vector machine(向量机), 55
 - VLIW(超长指令字计算机), 56
 - parallel processor(并行处理器), 58
 - array processors(阵列处理器), 58
 - MIMD(多指令流多数据流计算机), 58, 59
 - multiprocessor system(多处理器系统), 59
 - SIMD(单指令流多数据流计算机), 58
- Area(面积), 238, 274, 282, 298, 302
- Argos(一种图形同步语言), 99

B

- Basicblock(基本块), 6, 266, 303
- Behavior(行为), 65, 165
 - leaf(叶行为), 105, 122
- Behavioral decomposition, *see* Hierarchy, behavioral(行为

分解, 参见 Hierarchy, behavioral)

- Binary constraint search(二分约束搜索), 217
- Binding(绑定), 8, 279, 335
- Bitrate(位传输率), 244, 272
- Block diagram(框图), 10, 32, 377
- Branch probability(分支概率), 269
- BUD(一种系统设计系统), 205, 298
- Bus(总线), 313, 355
 - arbiter(总线仲裁器), 355
 - generation(总线生成), 315, 323
 - master(总线主端), 355
 - protocol, *see* Protocol(总线协议, 参见 Protocol)
 - rate(总线速率), 314, 316~319, 322
 - selection(总线选择), 359
 - slave(总线从端), 355
 - width(总线宽度), 314, 318~320, 322, 326

C

- Capture-and-simulate(捕获-模拟), 6, 9, 384, 386
- CCD(component-connectivity diagram)(组件连接图), 32
- CDFG(control/data flow graph)(控制/数据流图), 36
- CFG(control flow graph)(控制流图), 31
- Channel(通道), 78, 95, 152, 155, 244, 272, 288, 314
 - accesses(通道访问), 314
 - average rate(通道平均速率), 314, 316, 318
 - identification lines(通道标志线), 326, 327
 - peak rate(通道最高速率), 314, 318
 - size(通道大小), 314
- CHOP(一种系统设计技术), 213
- CISC(complex-instruction-set computer)(复杂指令集计算机), 51
- Clique partitioning(团划分), 275
- Clock cycle(时钟周期), 241, 251, 299, 302
- Clock slack minimization(最小时钟松弛), 254, 302
- Clock utilization(时钟利用), 254
- Closeness function(接近函数), 183
- Closeness metric(接近度量), 183, 223
- Clustering(结群), 187, 201, 205, 298
- Cluster tree(结群树), 189, 206, 209
- Communication(通信), 77, 91, 92, 94, 100, 108, 150,

152, 290
 broadcast(广播), 77, 99, 103
 channel *see* Channel(通信通道)
 message-passing(消息传递), 78, 82, 97, 108, 152
 shared-memory(共享存储器), 77, 152
 Compiler optimizations(编译优化), 295
 Completion(完成), 75, 88, 100, 101, 105, 109, 135
 Completion point(完成点), 75, 110, 123, 128
 Complex-instruction-set computer(复杂指令集计算机), 51
 Component implementation(组件实现), 381, 383
 Component-connectivity diagram(组件连接图), 32
 Conceptual model(概念模型), 15, 64, 65, 83, 108, 146 ~ 148, 164
 Conceptualization(概念化), 6, 145
 Conceptualization environment, *see* Environment(概念化环境, 参见 Environment)
 Concurrency(并发), 65, 66, 74, 88, 135
 bit-level(位级并发), 66
 control-driven(并发控制设计), 67
 data-driven(数据驱动并发), 66
 job-level(作业级并发), 66
 operation-level(操作级并发), 66
 statement-level(语句级并发), 66, 69, 89, 94, 159
 task-level(任务级并发), 66, 89, 94, 160
 Constraints(约束), 233
 Control access refinement(控制访问细化), 356, 366
 Control steps(控制步), 241, 260
 Control systems(控制系统), 21
 Control unit(控制单元), 249, 284
 Control-flow graph(控制流图), 31
 Control/data flow graph(控制/数据流图), 36, 178, 205, 210, 219, 389
 Controller(控制器), 47
 Controller architecture(控制体系结构), 47
 Cosimulation(协同模拟), 219
 Cost(成本), 182
 Cosyma(一种系统设计系统), 221
 CSP(通信顺序进程语言), 82, 84, 96
 Outline(切割线), 190

D

Data access refinement(数据访问细化), 356, 363
 Database(数据库), 385, 392, 397
 Dataflow(数据流), 66, 91, 92, 100, 102, 156, 178, 375
 Dataflow graph(数据流图), 29
 Datapath(数据通路), 48, 249, 275, 299

bit-sliced stack(位片堆栈), 282
 Datapath architecture(数据通路体系结构), 48
 Decomposition, *see* Hierarchy(分解, 参见 Hierarchy)
 Delta time(δ 时间), 167
 Describe-and-synthesize(描述-综合), 7 ~ 10, 384, 386
 Design domain(设计域), 17
 Design model(设计模型), 234, 237, 249
 Design process(设计过程), 1, 2, 6, 13, 17, 19, 60
 Design representation(设计表示), 1, 2
 behavioral(行为), 2 ~ 4
 physical(物理), 3, 4
 structural(结构), 3, 4
 Design time(设计时间), 248
 Design view(设计观点), 401
 DFG(Dataflow graph)(数据流图), 29

E

Embedded system(嵌入式系统), 13, 86, 104, 108, 133
 Entity-relationship diagram(实体关系图), 34
 Environment(环境), 385, 386, 397
 ERD(Entity-relationship diagram)(实体关系图), 34
 Esterel(一种系统设计系统), 103
 Estimation(评估), 12, 180, 233, 380, 387, 399
 accuracy(评估精度), 235
 fidelity(评估保真度), 181, 236
 hardware(硬件评估), 249, 302
 software(软件评估), 290, 303
 Event graph(事件图), 350
 Exceptions(异常, 例外), 83, 88, 92, 93, 103, 108, 127, 135, 161, 164
 Executable specification, *see* Specification, executable(可执行描述, 参见 Specification, executable)
 Execution frequency(执行频率), 269, 270
 Execution time(执行时间), 242, 268, 273, 295, 299
 Exploration(探索), 11 ~ 13

F

Finite-state machine(有限状态机), 20, 65, 72, 86, 97, 99, 100, 110, 162
 hierarchical(层次式有限状态机), 150
 state-based(基于状态的有限状态机), 22
 transition-based(基于状态迁移的有限状态机), 22
 Finite-state machine with datapath(带数据通路有限状态机), 23, 50
 Flow analysis(流程分析), 269, 273, 303
 Flow chart(流程图), 31
 Fork-join(分叉-连接), 68, 81, 92, 103, 112, 159, 164
 FPGA(现场可编程门阵列), 395

FSM(Finite-state machine)(有限状态机), 20
 FSMID(Finite-state machine with datapath)(带数据通路有限状态机), 23, 50, 178, 249
 Functional errors(功能错误), 132, 133, 136
 Functional objects(功能对象), 309, 381

G

Gate-level component(门级元件), 5, 9
 Gate-level schematic(门级原理图), 32
 Genetic evolution(遗传进化), 198
 Greedy algorithm(贪心算法), 184, 214
 Group migration(成组移植), 191

H

Hardware(硬件), 12, 354
 Hardware behavior(硬件行为), 357
 Hardware component(硬件组件), 355
 Hardware description language(硬件描述语言), 63, 65, 88
 Hardware partition(硬件划分), 356
 Hardware port(硬件端口), 357
 Hardware variable(硬件变量), 357
 Hardware/software partition, *see* Partitioning, hardware/soft(软硬件划分, 参见 Partitioning, hardware/soft)
 HardwareC, 69, 93
 HCFSM(层次并发有限状态机), 27
 Hierarchical concurrent finite-state machine(层次并发有限状态机), 27
 Hierarchy(层次), 70, 134, 135, 150, 158
 behavioral(层次式行为), 72, 74, 76, 87, 89, 92, 96, 98, 101, 103, 104, 121, 165
 concurrent decomposition(并发分解), 74, 98, 104, 126
 sequential decomposition(顺序分解), 72, 98, 104, 121~123, 127, 128, 134
 structural(层次式结构), 71, 89, 92, 93, 100, 103, 108, 158
 High-level synthesis, *see* Synthesis, chip(高层次综合, 参见 Synthesis, chip)
 Hill-climbing algorithm(爬山算法), 184, 216
 History(历史), 99

I

Integer linear programming(整数线性规划), 200, 225
 Interactive television(交互式电视), 375
 Interface(接口), 121, 393
 incompatibility(不相容接口), 336, 337
 process generation(接口进程生成), 342, 345, 349
 refinement(接口细化), 313
 Interface generation(接口生成), 356, 361, 362

Intermediate forms(中间格式), 386

Jackson's diagram(Jackson 图), 35

K

Kernighan/Lin(Kernighan/Lin), 191

L

Language(语言)
 application-specific(专用语言), 147
 front-end(前端语言), 147
 standard(标准语言), 145~147
 Left-edge algorithm(左边算法), 278
 Local minimum(局部最小), 184
 Logic minimization(逻辑最小化), 7, 393

M

Manual design(手工设计), 140, 386, 402, 403
 Manufacturing cost(制造成本), 248
 Memory address translation(存储器地址翻译), 312
 Message passing, *see* Communication, message-passing(信息传递, 参见 Communication, message-passing)
 Methodology(方法学), 6, 7, 10, 11, 13, 373
 Metric, *see* Quality metric(度量, 参见 Quality metric)
 Min-cut(最小割), 191
 Model(模型), 16~19
 activity-oriented(面向活动的模型), 19, 29, 31
 data-oriented(面向数据的模型), 20, 34, 35
 Heterogeneous(异构模型), 20, 36, 39, 40, 42, 43
 state-oriented(面向状态的模型), 19, 20, 24, 27
 structure-oriented(面向结构的模型), 20, 32
 tightly-integrated(紧密集成的模型), 20
 Multi-stage clustering(多级群集), 190, 210, 300

N

Natural language, *see* specification, natural-language(自然语言, 参见 Specification, natural-language)
 Non-determinism(非确定性), 83, 97

O

Object-oriented model(面向对象的模型), 42
 Objective function(目标函数), 182, 212, 220, 223
 Operator-use method(运算符-使用方法), 260

P

Partitioning(划分), 12, 171, 246, 382, 387
 abstraction level(抽象级别划分), 177
 algorithms(划分算法), 183, 186, 228
 granularity(划分粒度), 179, 223
 hardware/software(软硬件划分), 174, 175, 214, 221
 output(划分输出), 184

structural vs. functional(结构与功能划分), 172
 Performance(性能), 240, 300, 302
 Petri net(Petri 网), 24
 Physical design(物理设计), 395
 Pins(引脚), 239, 288, 303
 Pipelining(流水线), 243, 251, 262
 latency(时延), 243
 throughput(吞吐量), 243
 Power(功率), 247, 305
 Processor-level component(处理器级组件), 5
 Program-state machines(程序状态机), 43, 65, 70, 75, 83, 104, 121, 133, 136, 164
 Programming constructs(程序构造), 74, 88, 90, 92, 94, 96, 103, 134, 139
 Programming language paradigm(程序语言模式), 40
 Protocol(协议), 273, 314, 326, 327, 336, 338, 342
 delay(时延), 314, 324
 generation(协议生成), 315, 326, 329
 Protocol converter(协议转换器), 353
 PSM(Program-state machine)(程序状态机), 43

Q

Quality metric, *see* Estimation(质量度量, 参见 Estimation), 180, 223, 233, 238, 400, 403
 Queue(队列), 155, 159
 Queueing model(排队模型), 45

R

Ratio cut(比例切割), 194
 Reduced-instruction-set computer(精简指令集计算机), 53
 Refinement(细化), 12, 13, 309, 382
 channel groups(通道组), 313
 variable groups(变量组), 310
 Register-level component(寄存器级元件), 5, 8, 9
 Register-level schematic(寄存器级原理图), 32
 Regularity(规则), 228
 RISC(Reduced-instruction-set computer)(精简指令集计算机), 53

S

Scheduling(调度), 8, 264, 389
 force-directed(力指向调度), 279
 Scoping(范围), 165
 SDL(规范与描述语言), 100
 Send/receive(发送/接收), 78, 327
 blocking(阻塞式发送/接收), 79, 152
 nonblocking(非阻塞式发送接收), 79, 154
 Sequential statements(顺序语句), 128

Shared port(共享端口), 357
 Shared variable(共享变量), 357
 SIERA(一种系统设计环境), 353
 Silage(一种数字信号处理系统的描述语言), 101
 Simulated annealing(模拟退火), 196
 Simulation(模拟), 133, 381, 386
 Software(软件), 11, 354
 Software behavior(软件行为), 357
 Software component(软件组件), 355
 Software partition(软件划分), 356
 Software port(软件端口), 357
 Software size(软件大小), 239, 296, 297
 Software variable(软件变量), 357
 SpecCharts(SpecCharts 语言), 104, 121, 133, 136 ~ 138, 140, 164, 165, 302
 Specification(系统描述, 规范), 6, 10, 13, 15
 executable(可执行系统描述), 5, 10, 11, 64, 132, 133, 140, 145, 176, 381
 natural-language(自然语言系统描述), 15, 64, 132, 134, 140, 175, 379
 Specification Refinement, *see* Refinement(系统描述细化, 参见 Refinement)
 Specify-explore-refine(描述-探索-细化), 13
 SpecSyn(一种系统设计描述语言), 222, 302
 State encoding(状态编码), 393
 State minimization(状态最小化), 7, 393
 State transition, *see* Transition(状态迁移, 参见 Transition)
 Statecharts(一种状态图描述语言), 65, 81, 83, 97, 138, 139
 Structural decomposition, *see* Hierarchy, structural(结构分解, 参见 Hierarchy, structural)
 Structure chart(结构图), 39
 Synchronization(同步), 80, 91, 92, 95, 99, 101, 104, 107
 Synthesis(综合), 384, 402
 arbiter(仲裁器), 387
 behavioral(行为综合), 7, 9
 chip(芯片综合), 168, 385, 389
 high-level(高层次综合), 7
 interface(接口综合), 387, 394
 logic(逻辑综合), 6, 7, 9, 385, 393
 memory(存储器综合), 394
 software(软件综合), 385, 395
 system(系统综合), 385, 387
 System component(系统组件), 171, 180, 223, 309
 System design(系统设计), 15, 380 ~ 382

System functionality(系统功能), 1~3, 10, 15, 16, 18, 64, 133, 140, 145, 377, 379, 381

System implementation(系统实现), 2, 18

System specification, *see* Specification(系统描述, 参见 Specification)

T

Technology mapping(工艺映射), 7

Testability(可测试性), 247

Testbench(测试用例), 129

Time shift(时间推移), 167

Time to market(上市时间), 248

Timeout(超时), 84, 91, 99, 107, 108

Timing(时序, 时延), 84, 93, 101

constraints(时延约束), 85, 95, 246, 340

functional(功能时序), 84, 91, 99, 102, 108, 162

Tradeoffs(折中), 225

Transduce synthesis(转换器综合), 350

Transformation(变换), 387

Transistor-level component(晶体管级元件), 4

Transition(迁移, 转换), 69, 73, 86, 97, 99~101, 105, 107, 110, 123, 134, 149, 164

Immediately(TI)(即时迁移), 107, 110, 125, 128, 135

on completion(TOC)(迁移完成), 75, 105, 110, 123, 124

Translation(翻译), 147

U

Uninterpreted operations(非解释的操作), 158

User interface(用户接口), 399

V

Variable distribution(变量分配), 356, 357

Variable folding(变量折叠), 310

Variable lifetime(变量生存期), 275, 299, 302

Vector machine(向量机), 55

Verification(验证), 64

Verilog, 92

Very-long-instruction-word computer(超长指令字计算机), 56

VHDL(一种程序设计语言), 84, 88, 105, 109, 112, 136~138, 149, 162, 164, 165, 302, 386

VLIW(Very-long-instruction-word computer)(超长指令字计算机), 56

Vulcan(一种系统划分工具), 219, 301

Y

Yorktown silicon compiler(一种硅编译器), 201